A Cost-Aware Region-Level Data Placement Scheme for Hybrid Parallel I/O Systems

Shuibing He, Xian-He Sun, Bo Feng, Xin Huang, Kun Feng
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{she11, sun, bfeng5, xhuang30, kfeng1}@iit.edu

***Abstract*—Parallel I/O systems represent the most commonly used engineering solution to mitigate the performance mismatch between CPU and disk performance; however, parallel I/O systems are application dependent and may not work well for certain data access requests. New emerging solid state drives (SSD) are able to deliver better performance but incur a high monetary cost. While SSDs cannot always replace HDDs, the hybrid SSD-HDD approach uniquely addresses common performance issues in parallel I/O systems. The performance of hybrid SSD-HDD architecture depends on the utilization of the SSD and scheduling of data placement. In this paper, we propose a cost-aware region-level (CARL) data placement scheme for hybrid parallel I/O systems. CARL divides large files into several small regions, calculates the region costs according to the data access patterns, and selectively places regions with high access costs onto the SSD-based file servers. We have implemented CARL under MPI-IO and the PVFS2 parallel file system environment. Experimental results of representative benchmarks show that CARL is both feasible and able to improve I/O performance significantly.**

***Keywords*-Parallel I/O System; I/O Middleware; Solid State Drive**

I. Introduction

The advance of semiconductor technology has dramatically improved the performance of CPUs during the past three decades. However, I/O performance has not improved at the same rate. While processor speeds have increased nearly 50% each year, the access latency of a single hard disk drive (HDD) has only reduced by roughly 7% [1]. Moreover, many HPC applications in scientific computing fields are becoming increasingly data intensive. Table I shows the data requirements of several representative applications run at Argonne National Laboratory in 2011 [2]. The large data access requirements of these applications are putting unprecedented pressure on computer I/O systems to store data effectively. Data access has become the major performance bottleneck of many HPC applications.

Parallel I/O systems are an effective approach to meet the high I/O demands of data intensive applications. In HPC clusters, a parallel I/O system typically consists of several layers including applications, I/O middleware, parallel file systems (PFS), and storage systems. In general, a PFS, such as PVFS [3], Lustre [4], GPFS [5] and PanFS [6], stripes file data across a cluster of file (I/O) servers which allows data requests from a compute node to be served concurrently by multiple file servers. Thus, I/O system performance is significantly improved by exploiting parallelism with PFSs.

However, the performance of PFSs is impacted by application I/O characteristics [7]–[9]. PFSs favor some I/O patterns, but perform poorly for others. For example, while PFSs are effective to improve I/O system performance for large requests; they fail to perform well for small requests due to the lack of parallelism and increased access latency. Even worse, PFSs hardly provide any benefit for small random requests because of the mechanical nature of disk head movements in HDDs. Typically, many data-intensive applications issue non-uniform data requests to a large file. Request sizes can be large when accessing one chunk of the file and small at another; some chunks are accessed more frequently than others. For applications with varied I/O access patterns, PFSs usually exhibit poor performance.

New emerging storage technologies, such as flash memory based SSDs, provide a possible hardware solution that can revolutionize I/O system design. Unlike traditional HDDs, SSDs are built on semiconductor chips without any mechanical component [10]. SSDs are able to provide one order of magnitude higher performance than HDDs, and are an ideal storage medium for building high performance I/O systems. For example, to improve the performance of data intensive applications, San Diego Supercomputer Center has built a large SSD-based high-performance computing cluster, called Gordon [11]. Even considering the price-drop trend led by technology advance, the cost of building I/O systems completely based on SSDs probably will not be reduced due to increased storage capacity requirements. Therefore, we believe that building a hybrid parallel I/O system with a large number of HDD-based file servers (HServer) and a small number of SSD-based file servers (SServer) is a promising way to address the I/O access problem.

While the hybrid SSD-HDD approach is promising in newer I/O systems, how to efficiently place data in these hybrid I/O systems is challenging. In general, the storage space of SServers is smaller than that of HServers. Without considering the hardware resource characteristics, blindly placing data onto the SServers does not best serve data accesses performance. For example, if a small seldom accessed portion of data is placed onto the Servers, the overall I/O performance can hardly be improved because a large volume of data is still

978-1-4799-0898-1/13/$31.00 ©2013 IEEE

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.

TABLE I
DATA REQUIREMENTS FOR SELECT 2011 INCITE APPLICATIONS AT ARGONNE LEADERSHIP COMPUTING FACILITY OF ANL [2].

Project	**On-line data** (TBytes)	**Off-line data** (TBytes)
Combustion in Gaseous Mixtures	1	17
Protein Structure	1	2
Laser-Plasma Interactions	60	60
Type Ia Supernovae	75	300
Nuclear Structure and Reactions	6	15
Fast Neutron Reactors	100	100
Lattice Quantum Chromo dynamics	300	70
Fracture Behavior in Materials	12	72
Engineering Design of Fluid Systems	3	200
Multi-material Mixing	215	100
Turbulent Flows	10	20
Earthquake Wave Propagation	1000	1000
Fustion Reactor Design	50	100

served from the HServers. Moreover, since large requests usually lead to better I/O parallelism and there are more HServers than SServers, serving large requests from SServers will incur less performance gains, or even degrade I/O throughput. As data access patterns of clients can vary at different parts of a file thus generating a significant impact on I/O performance, ideal data placement in a hybrid parallel I/O system must consider the data access patterns from the clients side.

Currently, a lot of work has focused on the data placement policy in an SSD-based hybrid storage system [12]–[15]. These methods are very helpful, however to the best of our knowledge, most of the work is deployed in a single file server, and none of the existing schemes have focused on data placement optimization for parallel I/O systems in a global view.

In this paper, we propose a cost-aware region-level data placement scheme (CARL) for a hybrid parallel I/O system, where HServers and SServers are deployed. CARL selectively places critical data from a large file onto SServers and can be beneficial to various types of I/O patterns. In summary, this study makes the following contributions.

- We propose a data access cost model for parallel file systems, which can evaluate the access time of a request with different access patterns and on different storage media.
- We present a region-level data placement scheme based on the cost model, which divides files into regions and selectively places regions with high access cost gains onto the underlying SServers.
- We implement and integrate the cost-aware region-level data placement scheme into the MPI-IO library; thus providing transparent access to applications, and portability to many different parallel file systems.
- We evaluate CARL with the IOR benchmark suite. Experimental results show that I/O throughput is significantly improved.

The rest of this paper is organized as follows. Section II discusses the related work. Section III describes the design and implementation of CARL. Section IV evaluates the performance with the IOR benchmarks and highlights the improvements. Finally, section V concludes the paper.

II. RELATED WORK

In this section, we focus on previous work in three aspects: I/O request stream optimization, data placement in HDD-based storage system, and data placement in SSD-based storage system.

A. I/O Software Optimization Approaches

Numerous efforts have focused on reorganizing I/O requests to produce large continuous data accesses. A lot of work has been done at the I/O middleware layer, including data sieving [16], list I/O [17], datatype I/O [18], two-phase I/O [19], and collective I/O [16]. Data sieving [16] techniques integrate multiple noncontiguous small requests into a larger contiguous chunk, possibly fetched with additional data (hole). List I/O [17] and datatype I/O [18] allow users to merge multiple I/O requests with different patterns into a single I/O routine. While list I/O is used to handle more general data access cases, datatype I/O is designed to access data with certain regularity. Two-phase I/O [19] and collective I/O [16] are techniques proposed to rearrange concurrent I/O accesses among a group of processes.

B. Data placement in HDD-based storage system

Optimizing data placement is another effective approach to improve I/O performance. Parallel file systems usually provide several data placement policies for different I/O workloads [8]. Data partition [20], [21] and replication [22], [23] techniques are also commonly used to organize data layout on file servers consistent with I/O workloads. Workload studies have shown that data accesses for most scientific applications usually fall into several patterns [23], therefore data placement optimization has to rely on the prior knowledge of data access patterns. For example, our previous work [9] proposed a data replication scheme, which identifies data access patterns, and creates reorganized data replications for identified patterns with optimized data layouts based on access cost analysis.

C. Data placement in SSD-based storage system

As SSDs exhibit a clear advantage in performance over traditional HDDs, they are widely deployed in parallel I/O systems. However, most of these approaches are done on a single file server. Using SSDs as a cache of traditional HDDs is one popular method to improve I/O performance. SieveStore [24] captures the most popular blocks and places them onto an SSD. iTransformer [25] and iBridge [26] redirect the most random requests to the SSD to achieve cost-effective storage acceleration. SSD-based hybrid storage is another cost-effective method to make full use of the potential of SSDs. These methods integrate an SSD and a hard disk as one block device [14], [15]. I-CASH is a new hybrid storage architecture

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.

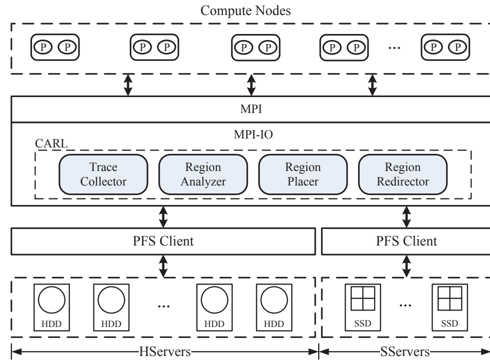


Fig. 1. Overview of system using CARL.

based on data-delta pairs which improves I/O performance for I/O intensive workloads [13]. Hystor identifies critical data blocks with strong temporal locality and redirects them to SSD for fast future accesses [12].

All the I/O request reorganization and data placement techniques are effective in improving parallel I/O performance. However, little effort has been done on data placement in a hybrid parallel I/O system deployed with both SSD-based and HDD-based file servers. In addition, our proposed scheme takes parallelism of parallel file system into account, and is another important difference from existing work focused on a single file server.

III. Cost-Aware Region-Level Data Placement

The proposed data placement scheme, CARL, aims to place critical data, which have high access costs, onto high-performance SSD-based file servers in order to improve the parallel I/O performance.

A. Overview of CARL

As data access patterns may vary at different parts of a file and make a significant impact on I/O performance, data placement in a hybrid parallel I/O system must consider the data access patterns from the client side. CARL divides a large file into several small regions and selectively places them onto the underlying file servers according to the data access patterns. This allows CARL to utilize the SSD and parallelism for all I/O requests.

Many data-intensive applications have regular patterns in accessing data files. These applications are often executed on a computer cluster many times, possibly with different sets of parameters. As the data access patterns are generally consistent from one run to another run, it provides an opportunity to optimize the data placement based on the I/O trace analysis.

Figure 1 shows the high performance cluster systems for which CARL is designed. In these systems, application processes on compute nodes access the data on the file servers by calling the MPI-IO library. CARL is resident in MPI-IO library and responsible for placing data onto the underlying HServers and SServers, which are accessed by a parallel file system respectively. Typically in such systems, there are a larger number of HServers and a small number of SServers with higher I/O performance and smaller storage space. CARL is independent of the file system; thus, allowing the scheme to be portable and easily adopted to multiple file systems, such as PVFS [3], Lustre [4], and GPFS [5].

Figure 2 shows the procedure of obtaining an optimal placement strategy with CARL, which consists of three phases. In the "Tracing Phase", the run-time statistics of data accesses are collected by the *trace collector* during the applications' first execution. In the "Analysis Phase", the *region analyzer* divides the file into regions and uses the data access cost model to estimate the data access performance gains for each region if they were placed on SServers over HServers. The performance gains are then used to generate a region gain table (RGT). In the "Placing Phase", the *region placer* places the file regions on the underlying file servers according to the RGT. To make our scheme transparent to the applications, the *region redirector* module is added at the I/O middleware layer (MPI-IO library) to forward I/O requests to the underlying HServers or SServers. Through these three phases, CARL reduces the I/O time for applications accessing files in similar patterns in later runs.

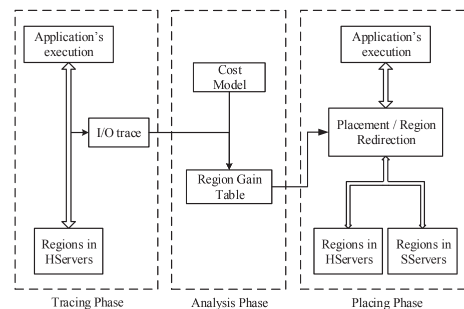


Fig. 2. The procedure of data placement scheme.

B. Trace Collecting

The *trace collector* is responsible for collecting run-time file access information of parallel applications. While there are some techniques and tools can be used for data analysis, we use IOSIG [27], which is an I/O pattern analysis tool developed in our previous work, to capture the information required by CARL. IOSIG is a pluggable library of MPI-IO, which uses the Profiling MPI interfaces (PMPI) to trace standard MPI-IO calls. 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.

C. Data Access Cost Model

In CARL, a model is proposed to calculate the data access time for each file request with corresponding parameters listed in Table II. By accumulating the cost of each file request on a file region, the cost of each region can be obtained.

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.

TABLE II
PARAMETERS (SHORT IN PARS) IN COST ANALYSIS MODEL.

Pars	Description
N	Number of HDD file servers
M	Number of SSD file servers ($M < N$)
Str	Stripe size of parallel file system
S	Data size of request req
a	Minimum startup time cost in HDD
b	Maximum startup time cost in HDD
β_H	HDD transfer time per unit data
β_S	SSD transfer time per unit data

For each file request req arriving at and served by HServers, the access cost is defined as

$$T_H = T_s + T_t \quad (1)$$

Here the cost is the completion time for each request, which consists two parts: startup time T_s and data transfer time T_t. The startup time means the time consumption due to disk seek and software overhead on the file servers. Data transfer time means the time spent on actual data read/write from/to HDD disk, and is proportional to the data size on the file servers. Assume the startup time on each file server is α, α is usually a random variable. We assume α follows uniform distribution on [a, b], then the probability function of α is

$$P(\alpha < x) = \frac{x-a}{b-a}, a \leqslant x \leqslant b \quad (2)$$

Because PFSs commonly stripe file data across multiple file servers, request req may involve several parallel data accesses on m file servers. According to the size of req, the value of m can fall into three cases, namely $m = 1$, $1 < m < N$, and $m = N$, as shown in Figure 3(a), (b), and(c) respectively. If a file request req accesses data on m file servers in parallel, the overall startup time should be the maximum of all the m file servers. Let α_1, α_2, $\cdots$, α_m be the startup time of the m file servers, $X = max(\alpha_1, \alpha_2, \cdots, \alpha_m)$, then the probability density function of X is

$$f(x) = \frac{m \times (x-a)^{m-1}}{(b-a)^m}, a \leqslant x \leqslant b \quad (3)$$

Hence, the expectation of the maximum startup time

$$\alpha = \int_a^b x f(x) dx = a + \frac{m}{m+1}(b-a) \quad (4)$$

On the other hand, the data transfer time T_t of request req should be the maximum of all m file servers, which is proportional to the data size in each file server. If $m = 1$, as shown in Figure 3(a), the maximum data size should be the request size S. Similarly, if $1 \leqslant m < N$, as shown in Figure 3(b), the maximum data size should be the stripe size Str. However, if $m = N$, as shown in Figure 3(c), the maximum data size cannot be obtained directly. In this case, let S_n denote the maximum data size, it can be calculated as

$$S_n = \lfloor \frac{S}{N * Str} \rfloor * Str + min\{Str, S\%(N * Str)\} \quad (5)$$

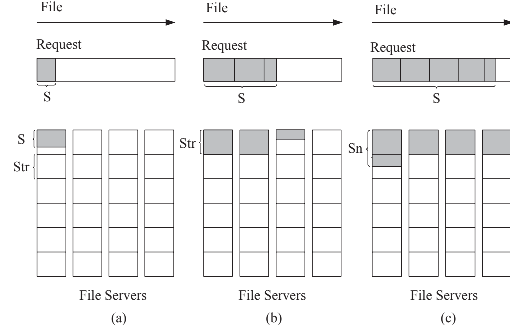


Fig. 3. Three cases where a file request involves a different number of file servers. (a) Only one file server is involved. (b) Three file servers are involved. (c) All the four file servers are involved.

Here S_n includes two parts as shown in the right side of Equation 5. $\lfloor \frac{S}{N*Str} \rfloor * Str$ is the data which is stripe-size-aligned and evenly divided by all the N file servers; $min\{Str, S\%(N * Str)\}$ is the overflowed data size.

Based on Equation 4 and 5, the data access cost of each file request can be expressed as the following three cases.

$$T_H = \begin{cases} \frac{1}{2}(a+b) + S * \beta_H, & S < Str \\ a + \frac{\lceil \frac{S}{Str} \rceil}{\lceil \frac{S}{Str} \rceil + 1} * (b-a) + Str * \beta_H, & Str \leqslant S < N * Str \\ a + \frac{N}{N+1} * (b-a) + S_n * \beta_H, & S \geqslant N * Str \end{cases} \quad (6)$$

In contrast, for request req served at SServers, we calculate the access cost without consideration of the seek time because SSDs are insensitive to spatial locality. Assume S_m is the maximum data size when all SSD file servers are involved in parallel data accesses, it can be defined as

$$S_m = \lfloor \frac{S}{M * Str} \rfloor * Str + min\{Str, S\%(M * Str)\} \quad (7)$$

Then the access cost for request req served by SServers is defined by

$$T_S = \begin{cases} S * \beta_S, & S < Str \\ Str * \beta_S, & Str \leqslant S < M * Str \\ S_m * \beta_S, & S \geqslant M * Str \end{cases} \quad (8)$$

D. Region Cost Analysis

With the proposed data access cost model, the performance gain of each file region by placing the region on SServers over HServers is calculated. The basic approach includes following three steps.

First, the address space of the file is logically divided into regions by a fixed chunk size (e.g. 32MB or 64MB) for further analysis.

Second, the I/O requests located on each region are identified according to the I/O traces. If the start offset of an I/O request falls into the region, the request is counted toward the region. If the request crosses several regions, then each subpart of the request contributes to the region it falls into.

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.

Third, the performance gain for each file region is estimated. Let $n(i)$ denote the number of requests located on the ith file region, T_H^j and T_S^j denote the data access cost time taken by HServers and SServers to serve the jth request respectively, which are calculated with Equation 6 and 8, then the gain g_i for the ith file region is defined by

$$g_i = \sum_{j=0}^{n(i)} (T_H^j - T_S^j) \tag{9}$$

After the above three steps, the region gains can be determined for a large file. These estimates are stored in a region gain table (*RGT*) to help make decisions in the data placement algorithm. Since *RGT* comprehensively considers the key factors in data accesses, such as number of requests, request frequency, request size, and the I/O parallelism of underlying file servers, it can be used to effectively guide the data placement.

E. Placement Algorithm

The *region placer* places data for each file region based on three factors: (1) the available free space in SServers, indicating whether SServers can accommodate the current region, (2) the value of the performance gain of the region, indicating whether I/O performance can be improved if it is placed on SServers, (3) the rank of the region performance gain, indicating whether it incurs more performance gains than other regions if it is located on SServers.

Algorithm 1 shows the algorithm of data placement for each I/O request. First, a global region map *RMT*, which keeps track of the location mapping information between logical file regions and target regions on HServers or SServers, is initialized. The *RMT* is empty at the beginning, and is continuously updated as new regions are allocated to the file. Upon each file request, the algorithm checks if the request falls into a region that has been allocated to the file by consulting the *RMT*. If so, the request is served with the allocated region. Otherwise, a new region from the underlying file servers is allocated to place the file data. Suppose there are n free regions on SServers currently, and the incoming request r belongs to file logical region reg, then the algorithm will allocate a target region from SServers for r only when both of the following conditions are true: (1) the performance gain of region reg is positive, (2) region reg is in the top-n of all the unallocated regions ordered by the performance gain. Otherwise, the algorithm will find a free region from HServers and place the data on the new allocated space.

Figure 4 shows an example of the proposed data placement scheme. In this example, the file is divided into five regions, where different regions have different access costs. Among the five file regions, region 2 and 4 have higher positive region gains than others. As there are two free regions on SServers, region 2 and 4 are placed onto SServers, and the rest are located onto HServers. Since the destination for each region is optimized according to the data access gains on them, the proposed data placement scheme can serve all I/O requests with high performance. The region-level data placement scheme is a fine-grained optimization, and it is more suitable for applications with complex data access patterns.

Algorithm 1 The region-level data placement algorithm

Require: I/O Request: r, Region gain table: RGT, Region map table: RMT.

```
1:  /* Lookup r in RMT, return a mapping entry reg*/
2:  reg ← RMT_lookup(r)
3:  if reg != NULL then
4:      if reg.position == SServers then
5:          Forward r to reg on SServers
6:      else
7:          Forward r to reg on HServers
8:      end if
9:  else
10:     /* Otherwise, place data to a new region */
11:     c ← Calculate the free capacity of SServers
12:     Let n = c/region_size
13:     /* Find top n regions in RGT but not in RMT */
14:     Reg[n] ← top_n({x : x ∈ RGT ∧ x ∉ RMT})
15:     /* Find a matched region in Reg[n] */
16:     for each reg in Reg[n] do
17:         if r in reg and reg.gain > 0 then
18:             reg ← Allocate a region from SServers
19:             Forward r to reg on SServers
20:         end if
21:     end for
22:     if no matched region found in Reg[n] then
23:         reg ← Allocate a region from HServers
24:         Forward r to reg on HServers
25:     end if
26:     Add an entry of reg into RMT
27: end if
```

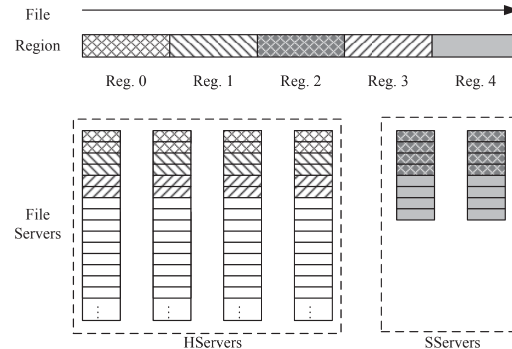


Fig. 4. An instance of the region-based data placement scheme.

F. Region Redirection

The *region redirector* in the MPI-IO library is responsible for redirecting user's I/O requests to the underlying HServers or SServers. In order to keep track of the location of each file region, the mapping table *RMT* is stored in a file in the same directory as the MPI program. The mapping table entries

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.

are also hashed in memory for efficient table lookup. Changes to the mapping entries in memory are synchronously written to the storage in order to survive power failures. We modify the MPI library so that the mapping table is loaded with MPI_Init() and unloaded with MPI_Finalize(). We also modify the MPI_File_read/write() (and other variants of read/write), so that the user requests can be atomically forwarded to the alternative file servers. In more detail, if the requested regions are found in *RMT*, the logical file regions will be translated to the target regions. Then, the following read/write operations will be forwarded to the target regions on underlying file servers. All the operations are transparent to applications. In this way, the SServers, which have a limited storage space and small number of file servers, can be intelligently utilized according to the I/O patterns.

IV. Performance Evaluation

In this section, we evaluate the performance of CARL through several benchmark-driven experiments. Before discussing the experiment results, we will first describe the experimental setup.

A. Experimental Setup

The experiments were conducted on a 65-node SUN Fire Linux cluster. Each computing node has two AMD Opteron processors, 8GB memory and a 250GB HDD (SEAGATE ST32502NSSUN250G). The operating system is Ubuntu 9.04 and the parallel file system is PVFS2 version 2.8.1. All nodes are equipped with Gigabit Ethernet interconnection, and 17 nodes are equipped with additional PCI-E X4 100GB SSD (OCZ-REVODRIVE X2).

Of the 20 standalone configured nodes, eight nodes are computing nodes, eight nodes are HServers, and four nodes are SServers. Both SServers and HServers are accessed through a PVFS2 parallel file system respectively. By default, data is striped over the file servers with a 64KB striping unit size. In general, the larger the capacity of an SServer; the better the I/O performance. In order to avoid overestimating the improvement, the data size on SServers is set to 20% of the application file size. To make our comparison fair and conservative, the operating system buffer caches are flushed before each run to ensure that all data will be read from the storage devices. In addition, the dirty data in the memory is periodically flush to the storage devices to ensure that write throughput on the storage devices is correctly measured.

IOR [28] is used to benchmark system performance. In order to show the effectiveness of CARL for different I/O patterns, IOR is used to generate two kinds of random workloads. One workload generates a uniform random distribution, e.g. the default implementation in IOR. The other generates a Zipfian distribution, and is implemented for this study. Since CARL tries to utilize the skewness in workloads during data placement, uniform random access is the workload that shows the worst case behavior of CARL.

For each workload, the baseline experiments on the HDD-only system show the original I/O performance. In order to show the effectiveness of CARL, experiments are conducted on the hybrid I/O system with a simple data placement scheme (RANDOM). RANDOM distributes regions of a large file randomly to the SServers.

B. Results and Analysis

1) Varying Request sizes: The IOR benchmark is executed with request sizes of 8KB, 16KB, 32KB, and 256KB. The number of processes n is fixed to 32. Each process is responsible for accessing its own $1/n$ of a 10GB shared file, and continuously issues requests with random offsets.

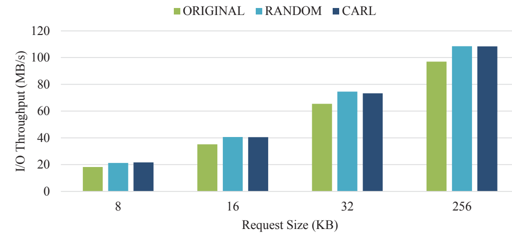


Fig. 5. I/O throughputs with varied request sizes for uniform random workload.

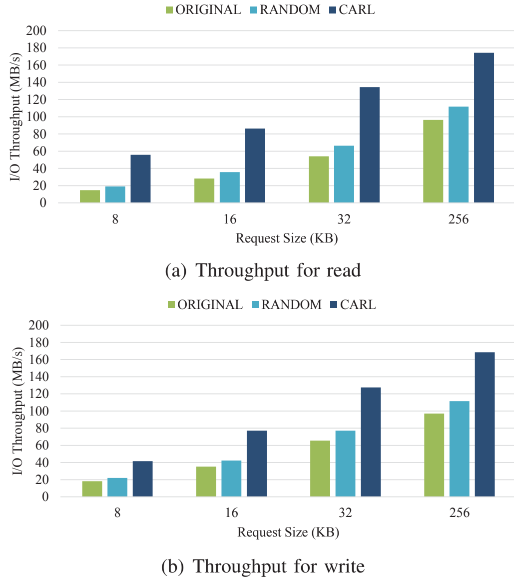
Figure 5 shows the read performance comparison for the uniform workload. It is observed that both CARL and RANDOM can improve the original I/O throughput by adding SSDs to the parallel I/O system. CARL improves the performance by 18.8%, 15.3%, 12.1% and 11.7%. With larger request sizes, the I/O throughput improves because each request is continuously served in HServers for a longer time. Compared with RANDOM, CARL has a small performance difference. This is because with uniform workloads, the regions selected by CARL nearly bring the same performance gain as RANDOM. In this case, CARL nearly degrades to RANDOM. Due to space constraints, only results for read test are presented here, the write test shows similar results.

Figure 6(a) shows the read performance for the Zipfian workload. CARL can improve the I/O performance by 278.7%, 205.1%, 148.4% and 80.9%, while RANDOM only improves the I/O performance by 29.3%, 26.1%, 22.6% and 16.0%. These results show that, as the request size increases, both CARL and RANDOM provide better performance compared to the results for a uniform workload. However, CARL has a significant performance improvement over RANDOM. This is because the most critical data, which are a few regions of files with the highest cost, are placed onto the SSDs; thus, the I/O access time is largely reduced.

The write test yields similar result as shown in Figure 6(b). In comparison to the baseline, CARL increases the throughput by 127.4%, 118.9%, 94.8% and 73.7%, respectively. Compared to RANDOM, CARL shows 88.1%, 82.3%, 65.6% and 51.0% improvements. However, CARL provides a more modest improvement in writes due the physical characteristics of SSDs which favor reads over writes.

2) Varying Number of Processes: The impacts on I/O performance are evaluated while varying the number of processes.

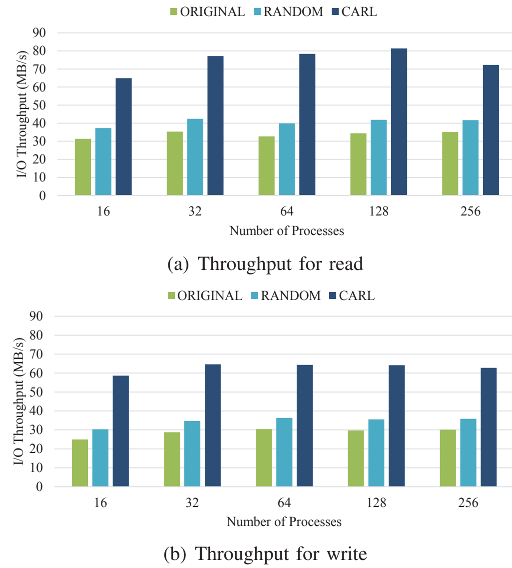
Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.



(a) Throughput for read

(b) Throughput for write

Fig. 6. I/O throughputs with varied request sizes for Zipfian random workload.



(a) Throughput for read

(b) Throughput for write

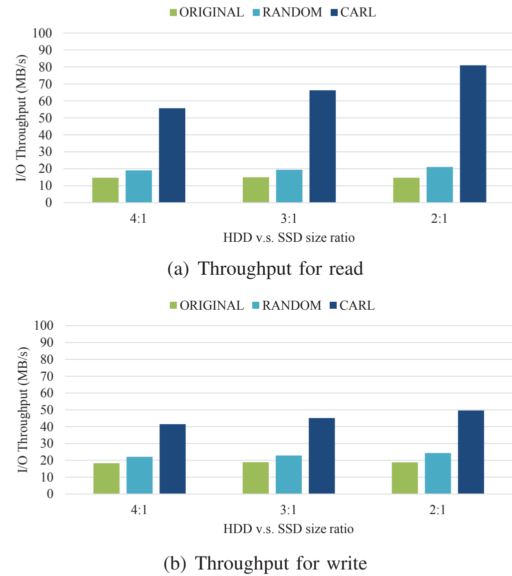
Fig. 7. I/O throughputs with varied number of processes for Zipfian random workload.

The IOR benchmark is executed with 16, 32, 64, 128 and 256 processes. The request size is fixed to 16KB.

Figure 7(a) shows the results of read performance for the Zipfian workloads. Similar to the previous test, the overall bandwidth is improved by between 107.2% to 139.7% through using CARL. With the number of processes increasing, the I/O bandwidth first increases and then decreases because each HDD file server needs to serve more process requests and the competition among processes impedes progress. Figure 7(a) shows another improvement of CARL: when the process number increases, the performance gain of CARL increases as well. In other words, CARL has better scalability and can handle more concurrent I/O processes. Additionally, the figure shows that CARL is more effective than RANDOM, and shows performance improvements of 74.2%, 81.9%, 96.7%, 94.5% and 73.2%. The performance trend is similar for write requests, as shown in Figure 7(b).

3) Varying SSD Sizes: In general, the capacity of SServers is much smaller than that of HServers and could be smaller than the I/O working set size for the application. In order to show the impacts of I/O performance due to SSD space, the I/O performance is evaluated with data size ratios of 4:1, 3:1, to 2:1 between HServers and SServers.

Figure 8(a) shows the I/O throughput for read operations. Similar to previous results, these results show that CARL is better than RANDOM, and has a performance improvement of 278.7%, 356.4% and 450.8% over the original I/O performance. I/O throughput improves by increasing the size of SServers. With the size of SServers increasing, the IOR bandwidth improves because more high-cost data regions are located on the SServers. However, the improvement has a limit. When most high-cost regions are already stored on SServers, enlarging SServers will not improve performance significantly. This is observed in Figure 8(a).



(a) Throughput for read

(b) Throughput for write

Fig. 8. I/O throughputs with varied SSD sizes for Zipfian random workload.

For writes, the I/O throughput is lower than reads, due to the better random read latency of SSD, but it also has a plateau, as shown in Figure 8(b).

V. Conclusions

Parallel I/O systems are widely used to mask the huge performance gap between computing and data accesses. However, they may exhibit poor performance for certain I/O patterns. In data-intensive parallel I/O environments, a single file could reach several terabytes, and data access patterns of the file could vary in different chunks of the file. Newer solid state

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.

drives (SSD) provide a possible hardware solution to the I/O system bottleneck. Due to the excellent performance but high cost of SSD, building parallel I/O systems with hybrid SSD-HDD file server is a promising way to address the I/O performance issue. In this paper, we proposed CARL, a cost-aware region-level data placement scheme to speed up the I/O performance for hybrid parallel I/O systems. This strategy provides fine-grained region-level data placement optimization, which is highly suitable for applications with non-uniform data access patterns.

We described the three-phase approach of the proposed data placement optimization, including how to divide the file into regions, how to calculate the performance gain for each region, and how to decide which regions should be placed on which file servers. The data placement scheme intelligently selects the proper underlying file servers for different file regions. It achieves a better integration of data access characteristics of applications and data organization on heterogeneous storage media in parallel file systems. The experimental results demonstrate that, the proposed data placement strategy with access cost awareness improves the performance up to 450.8% for reads and 167.3% for writes with the widely-used IOR benchmark.

In the future, we plan to enhance our cost model to include more complex data access patterns, and explore more optimization techniques to make I/O systems more intelligent and efficient.

Acknowledgment

This research was supported in part by National Science Foundation under NSF grant CNS-0751200, CCF-0937877 and CNS-1162540.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

[2] T. Peterka, “Parallel I/O in Practice,” in *CScADS Summer Workshops*, Snowbird, UT, USA, 2012.

[3] P. H. Carns, I. Walter B. Ligon, R. B. Ross, and R. Thakur, “PVFS: A Parallel Virtual File System for Linux Clusters,” in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.

[4] S. Microsystems, “Lustre File System: High-performance Storage Architecture and Scalable Cluster File System,” Tech. Rep. Lustre File System White Paper, 2007.

[5] F. Schmuck and R. Haskin, “GPFS: A shared-disk File System for Large Computing Clusters,” in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002, pp. 231–244.

[6] D. Nagle, D. Serenyi, and D. Serenyi, “The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage,” in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[7] 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 *Proceedings of the 11th IEEE/ACM International Symposium onCluster, Cloud and Grid Computing (CCGrid)*, 2011, pp. 414–423.

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

[9] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur, “Pattern-Direct and Layout-Aware Replication Scheme for Parallel I/O Systems,” in *Proceedings of 27th IEEE International Parallel and Distributed Processing Symposium*, 2013.

[10] F. Chen, D. A. Koufaty, and X. Zhang, “Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives,” in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, 2009, pp. 181–192.

[11] A. Caulfield, L. Grupp, and S. Swanson, “Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications,” in *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’09)*, 2009.

[12] F. Chen, D. A. Koufaty, and X. Zhang, “Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems,” in *Proceedings of the international conference on Supercomputing*, 2011, pp. 22–32.

[13] Q. Yang and J. Ren, “I-CASH: Intelligently Coupled Array of SSD and HDD,” in *Proceedings of the IEEE 17th International Symposium on High PerformanceComputer Architecture*, 2011, pp. 278–289.

[14] H. Payer, M. Sanvido, Z. Bandic, and C. Kirsch, “Combo Drive: Optimizing Cost and Performance in a Heterogeneous Storage Device,” in *Proceedings of the First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, vol. 1, 2009, pp. 1–8.

[15] T. Bisson and S. A. Brandt, “Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests,” in *Proceedings of the 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007, pp. 402–409.

[16] R. Thakur, W. Gropp, and E. Lusk, “Data Sieving and Collective I/O in ROMIO,” in *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp. 182–189.

[17] A. Ching, A. Choudhary, K. Coloma, L. Wei-keng, R. Ross, and W. Gropp, “Noncontiguous I/O Accesses through MPI-IO,” in *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003, pp. 104–111.

[18] A. Ching, A. Choudhary, W.-k. Liao, R. Ross, and W. Gropp, “Efficient Structured Data Access in Parallel File Systems,” in *Proceedings of the IEEE International Conference on Cluster Computing*, 2003, pp. 326–335.

[19] R. Thakur and A. Choudhary, “An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays,” *Scientific Programming*, vol. 5, no. 4, pp. 301–317, 1996.

[20] Y. Wang and D. Kaeli, “Profile-Guided I/O Partitioning,” in *Proceedings of the 17th Annual International Conference on Supercomputing*. San Francisco, CA, USA: ACM, 2003, pp. 252–260.

[21] S. Rubin, R. Bodk, and T. Chilimbi, “An Efficient Profile-Analysis Framework for Data-Layout Optimizations,” *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 140–153, 2002.

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

[23] 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*. New York, NY, USA: ACM, 2005, pp. 263–276.

[24] T. Pritchett and M. Thottethodi, “SieveStore: a Highly-Selective, Ensemble-level Disk Cache for Cost-Performance,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 163–174.

[25] X. Zhang, K. Davis, and S. Jiang, “iTransformer: Using SSD to Improve Disk Scheduling for High-performance I/O,” in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium*, 2012, pp. 715–726.

[26] X. Zhang, K. Liu, K. Davis, and S. Jiang, “iBridge: Improving Unaligned Parallel File Access with Solid-State Drives,” 2013.

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

[28] “Interleaved Or Random (IOR) Benchmarks.” [Online]. Available: http://sourceforge.net/projects/ior-sio/

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:47:09 UTC from IEEE Xplore. Restrictions apply.