Cost-Aware Region-Level Data Placement in Multi-Tiered Parallel I/O Systems

Shuibing He, Yang Wang, Zheng Li, Xian-He Sun *IEEE Fellow*, and Chenzhong Xu *IEEE Fellow*

Abstract— Multi-tiered Parallel I/O systems that combine traditional HDDs with emerging SSDs mitigate the cost burden of SSDs while benefiting from their superior I/O performance. While a multi-tiered parallel I/O system is promising for data-intensive applications in high-performance (HPC) domains, placing data on each tier of the system to achieve high I/O performance remains a challenge. In this paper, we propose a cost-aware region-level (CARL) data placement scheme in multi-tiered parallel I/O systems. CARL divides a large file into several small regions, and then places regions on different types of servers based on region access costs. CARL includes a static policy S-CARL and a dynamic policy D-CARL. For applications whose I/O access patterns are completely known, S-CARL calculates the region costs within the entire workload duration, and uses a static data placement scheme to selectively place regions on the proper servers. To adapt to applications whose access patterns are unknown in advance, D-CARL uses a dynamic data placement scheme which migrates data among different servers within each time window. We have implemented CARL under MPI-IO library and OrangeFS parallel file system environment. Our evaluation with representative benchmarks and an application shows that CARL is both feasible and able to improve I/O performance significantly.

Index Terms—Parallel I/O System; Parallel File system; Data Placement; Solid State Drive

1 Introduction

TODAY many of the applications in high-performance computing (HPC) domains are becoming increasingly data-intensive [1]. To satisfy the huge data requirements of such applications, HPC clusters use parallel I/O systems, which integrate multiple servers with a parallel file system(PFS) [2]–[5], to provide efficient storage accesses. However, the performance of PFSs is still severely impacted by application I/O characteristics [6]–[8]. For example, although PFSs are effective to improve I/O system performance for large requests, they fail to perform well for non-contiguous small requests. Therefore, a large body of studies are devoted to improve parallel I/O system performance [9]–[11].

Newly emerging storage technologies, such as flash-based solid state drives (SSD), are becoming increasingly popular in I/O system designs. When compared to hard disk drives (HDD), SSDs have higher storage density, lower energy consumption, a smaller thermal footprint, and orders of magnitude higher performance [12]. SSD is an ideal storage medium for building high performance I/O systems [13]. However, the high price per gigabyte of SSDs prevents them from being utilized to build an I/O system completely based on SSDs [14]. Hence, a multi-tiered parallel I/O system, which consists of both HDD-based file servers (HServer) and SSD-based file servers (SServer), is one of the practical ways to address the I/O bottleneck problem [15]–[18].

While a multi-tied HDD-SSD architecture is cost effective, the performance of the multi-tiered I/O system relies on an efficient data placement scheme. However, I/O access patterns and storage system configurations become more and more complex, how to place data in a multi-tiered parallel I/O system is challenging.

First, complicated I/O access patterns may result in inefficient data placement. A naive data placement approach is to place performance-critical data on SServers. For example, small requests can benefits more from SSDs, hence it tends to place data with small requests on SServers. However, previous studies have shown that applications can send I/O requests with complicated access patterns [7], [19], in terms of request size, type, frequency, and concurrency. A given data placement scheme can benefit requests with one given access pattern, but not necessarily lead to the optimal performance for other patterns. If we blindly place data on a tier without carefully considering the I/O access characteristics, the overall I/O system performance will be degraded.

Second, storage system configurations can also affect the efficiency of the data placement schemes. Generally, multi-tier parallel I/O systems may have different sys-

- *Shuibing He is with the State Key Laboratory of Software Engineering, Computer School, Wuhan University, Luojiashan, Wuhan, Hubei, China 430072, and Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Xueyuan Blvd. 1068, Shenzhen, China 518055 E-mail: heshuibing@whu.edu.cn*
- *Yang Wang is with Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Xueyuan Blvd. 1068, Shenzhen, China 518055. E-mail: yang.wang1@siat.ac.cn*
- *Zheng Li is with School of Computer Sciences, Western Illinois University, Macomb, Illinois, USA 61455. E-mail: z-li2@wiu.edu*
- *Xian-He Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois, USA 60616. E-mail: sun@iit.edu*
- *Chengzhong Xu is with the Cloud Computing Center, Shenzhen Institute of Advanced Technology, Chinese Academy of Science, 518055, Shenzhen, China and the Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202. E-mail: cz.xu@siat.ac.cn.*

tem configurations in terms of server performance and number of servers in each storage tier. A data placement policy works well under a special system configuration does not yield performance benefits for other configurations. For example, placing the requested data of a large file request on SServers is favorable if SServers have higher aggregated I/O performance than HServers. However, when a multi-tiered system has many more HServers than SServers, it is better to place data on HServers because of their higher I/O parallelism. As a result, an ideal data placement for a multi-tiered parallel I/O system must consider storage system configurations to determine the proper placement of file data.

Currently, plenty of work has been done on data placement policies in SSD-based hybrid I/O systems [14], [20]–[22]. These methods are very helpful, however, to the best of our knowledge, the existing work is deployed in a single file server, without considering data placement optimization in a multi-tiered parallel I/O system.

In this paper, we propose a cost-aware region-level data placement scheme (CARL) for a multi-tiered parallel I/O system combining both HServers and SServers. The basic idea of CARL is to divide a large file into several small regions, and then places file regions on different types of file servers based on the region access costs. By selectively placing fine-grained region instead of the entire file on the proper server tier, CARL can benefit various I/O patterns and system configurations. CARL consists of two data placement polices for different applications. First, for applications whose I/O access patterns are completely known, CARL calculates the region costs according to data access patterns within the entire workload duration, and uses a static data placement scheme to selectively place file regions with high access costs on proper servers with better aggregated I/O performance. Second, CARL also utilizes a dynamic data placement scheme which leverages data migration to place data on different types of servers based on workload changes if we have no a prior knowledge about the application's access patterns. As opposed to the static data placement scheme [15], such a dynamic data placement is more realistic and can adapt to applications with unknown access patterns.

Specifically, this study makes the following contributions.

- We introduce 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.
- For applications with I/O access patterns are completely known in advance, we present a static region-level data placement scheme based on the cost model, which divides files into regions and selectively places regions on proper underlying servers.
- For applications whose I/O access patterns are unknown, we propose a dynamic region-level data placement scheme, which considers data migration among different types of servers based on the region access costs.
- We implement and integrate the cost-aware region-level data placement scheme into MPI-IO library and OrangeFS, and evaluate the performance of CARL with extensive tests. Experimental results show that CARL can significantly improve the multi-tiered I/O system performance.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the data access cost model used in the proposed data placement scheme. Section 4 and Section 5 describe the static and dynamic region-level data placement policy respectively. Section 6 evaluates the performance of CARL. Finally, section 7 concludes the paper.

2 Related Work

In this section, we focus on previous studies in improving parallel I/O system performance: I/O software optimization and data placement in homogeneous/heterogeneous I/O systems.

2.1 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 [10], list I/O [11], datatype I/O [23], two-phase I/O [24], and collective I/O [10]. Data sieving [10] techniques integrate multiple noncontiguous small requests into a larger contiguous chunk, possibly fetched with additional data (hole). List I/O [11] and datatype I/O [23] 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 [24] and collective I/O [10] are proposed to rearrange concurrent I/O accesses among a group of processes.

2.2 Data placement in Homogeneous I/O systems

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 [6]. Data partition [25], [26], data migration [27], [28], and data replication [6], [8], [29], [30] techniques are commonly used to organize data layout on file servers consistent with I/O workloads. Furthermore, file stripe resizing technique is widely used to optimize the data placement of parallel I/O systems [7], [31]. PARLO is designed for accelerating queries on scientific datasets by applying user specified data placement optimizations [32]. Tantisiriroj *et al.* [33] use HDFS-specific layout rearrangement to improve the performance of PVFS [34]. However, all these studies are designed for homogeneous I/O systems, and cannot be applied to heterogeneous environments.

2.3 Data placement in Heterogeneous I/O systems

As SSDs exhibit obvious performance advantages over HDDs, they are widely deployed in parallel I/O systems, either as a cache of traditional HDDs [35], [36] or as a hybrid storage device [14], [18], [20], [21]. However, most of these approaches are made on a single file server. In contrast to these studies, our work is designed for a parallel I/O system.

Previous studies [37]–[41] use the global data information and SSDs in a similar way to optimize data placement in a parallel I/O environment. However, both SSD-based servers and HDD-based servers are used as persistent storage and the system only includes one storage tier. While recent studies also focus on the data placement in a multi-tiered parallel I/O system [9], [42], they use SSD-based server as a caching tier as apposed to our work that uses them as a storage tire, one of promising approaches to utilize high-performance SSDs

TABLE 1: Parameters (Pars in short) in cost analysis model.

Pars	Description
M	Number of HDD servers
N	Number of SSD servers
str	Stripe size of parallel file system
o	Offset of request req
s	Size of request req
p	Number of processes of the parallel application
α_h	Average startup time on HServer
β_h	Unit data transfer time on HServer
α_{sr}	Average startup time for read on SServer
β_{sr}	Unit data transfer time for read on SServer
α_{sw}	Average startup time for write on SServer
β_{sw}	Unit data transfer time for write on SServer

3 Data Access Cost Model

To guide the region-level data placement, we propose a cost model to calculate the data access time on a parallel file system. The corresponding parameters are listed in Table 1. The model not only considers application's access patterns (e.g., request size, offset, number of processes), but also takes storage system configurations (e.g., number of servers, server type, storage startup time and storage transfer time of each server) into account to overcome the challenges in data placements in multi-tiered parallel I/O systems.

In the cost model, we assume a file request is served either by HSevers or SServers, each organized by a parallel file system (PFS) in a multi-tiered I/O system. We also assume the file is distributed on the underlying servers in a round-robin fashion which is the most popular data layout method in a PFS [8]. We calculate the data access cost on different types of servers respectively as follows.

3.1 Data Access Cost on HServers

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

$$T_H = T_{hs} + T_{ht} \tag{1}$$

The cost is the completion time for each file request, which consists of two parts: storage startup time T_{hs} and storage transfer time T_{ht}. The storage startup time means the time consumption due to disk seek and software overhead on the file servers. Storage transfer time means the time spent on actual data read/write from/to an HDD disk.

3.1.1 *Storage Startup Time*

A parallel file request req may involve multiple sub-requests on m $(1 \leqslant m \leqslant M)$ HServers, the startup time of req is determined by the maximum of all its sub-requests. We first calculate the startup time of a single sub-request, then describe the startup time of the entire file request.

Let α denote the startup time of a sub-request on a single HServer, then α mainly depends on the number of seeks on the disk. Because an HServer will randomly serve sub-requests from multiple processes of an application [43], α is a random variable. Assume α follows a uniform distribution on [a, b], then its probability function is

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

where a and b are the minimal and maximal startup time on an HServer, respectively.

Let X denote the startup time of the entire file request req, then we have$X = \max(\alpha_1, \alpha_2, \cdots, \alpha_m)$, where α_i ($1 \leqslant i \leqslant m$) has an independent identical distribution as α. The probability density function of X is

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

Hence, the startup time of the entire file request is

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

An HServer only needs one seek operation to serve the sub-requests in the best case, thus $a = \alpha_h$. But in the worst case, there are p seeks since an HServer has to concurrently serve p processes, thus $b = p * \alpha_h$. Given a file request req with offset o and size s, the serial number of the involved beginning and ending stripe are $B = \lfloor \frac{o}{str} \rfloor$ and $E = \lfloor \frac{o+s}{str} \rfloor$. Let $c = E - B + 1$, thus

$$m = \begin{cases} c, & \text{if } c < M \\ M, & otherwise \end{cases} \tag{5}$$

Based on the value of a, b, and m, we can obtain the cost of T_{hs} according to Equation (4).

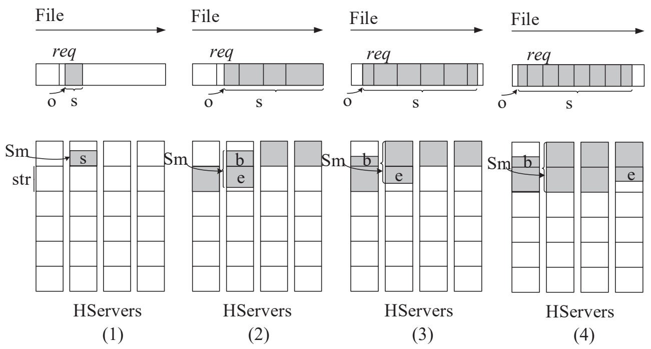


Fig. 1: Four cases where a file request involves different number of sub-requests.

3.1.2 Storage Transfer Time

The storage transfer time T_{ht} of request req should be the maximum of all its m sub-requests. Since each sub-request's data transfer time is proportional to the data size on the HServer, we first calculate the data size of each sub-request, then describe T_t for the entire file request based on the maximal sub-request size.

Given a file request req with offset o and size s, the size of the beginning and ending stripe fragment can be calculated as $b = str - o\%str$ and $e = (o+s)\%str$, as shown in Fig. 1. Let $r = \lceil \frac{E-B}{M} \rceil - 1$ and $s(i)$ be the sub-request size on server i $(1 \leq i \leq m)$, then $s_m = \max\{s(1), s(2), ..., s(m)\}$ can be calculated as follows:

$$s_m = \begin{cases} s, & c = 1 \\ \max\{b+e, str\} + r * str, & (c-1)\%M = 0 \\ \max\{b, e\} + r * str, & (c-1)\%M = 1 \\ (r+1) * str, & otherwise \end{cases} \quad (6)$$

Based on the value of s_m, the storage transfer time can be calculated as

$$T_{ht} = s_m * \beta_h \quad (7)$$

With Equation (4) and Equation (7), T_H of each file request in Equation (1) can be obtained.

3.2 Data Access Cost on SServers

For each file request served by SServers, we calculate its access cost in a similar way as that of HServers but with two modifications. First, we count request type (read or write) in the cost since SSDs usually have a much lower write performance because of garbage collection and wear leveling [12]. Second, we use sub-request distribution on SServers rather than HServers to derive the cost. The access cost is defined as

$$T_S = T_{ss} + T_{st} \quad (8)$$

where T_{ss} means the storage startup time and T_{st} refers to the storage transfer time on SServers.

3.2.1 Storage Startup Time

We assume that n $(1 \leqslant n \leqslant N)$ is the number of involved SServers in the data access of file request req, then n can be calculated similarly as m discussed in Subsection 3.1.1. Due to space limitation we omit the calculation here. Based on the value of n, the storage startup time on SServers can be calculated as

$$T_{ss} = a + \frac{n}{n+1}(b-a) \quad (9)$$

where a and b are the minimal and maximal startup time on an SServer, respectively.

If req is a read request, then $a = \alpha_{sr}$ and $b = p * \alpha_{sr}$. Here we set a and b with these values because each SServer only needs one seek operation to serve a continuous request in the best case and needs p startup operations to concurrently serve all the p processes in the worst case. Otherwise, $a = \alpha_{sw}$ and $b = p * \alpha_{sw}$ for a write request.

3.2.2 Storage Transfer Time

The storage transfer time T_{st} of a request is the maximal transfer time of all its n sub-requests. Given a file request req with offset o and size s, the maximal sub-request size can be calculated similarly as that of maximal sub-request size on HServers, as discussed in Subsection 3.1.2. Let s_n be the maximal sub-request size, then T_{st} can be calculated as follows:

$$T_{st} = \begin{cases} s_n * \beta_{sr}, & \text{if } req \text{ is a read} \\ s_n * \beta_{sw}, & otherwise \end{cases} \quad (10)$$

With Equation (9) and Equation (10), T_S of each file request in Equation (8) can be obtained.

3.3 Discussion

The proposed model is based on our previous work [15], but there are three major differences. First, from the application's perspective, our model considers the number of processes, which affects the storage startup time of a file request. Second, from the viewpoint of storage, our model also factors data startup time in the data access times of SServers while the previous model is an ideal case without such consideration. Third, our model differs read from write performance of SServers while the previous work regards them as the same. By considering these differences, our model can more accurately describe the performance of a practical I/O system.

4 Static Region-Level Data Placement

The proposed cost model can be used to determine which type of servers is the proper storage tier for a given file request. If we have a prior knowledge of all requests on a large parallel file, we can determine which parts of a file should be placed on which type of servers to achieve optimal I/O performance. Fortunately,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2636837, IEEE Transactions on Parallel and Distributed Systems

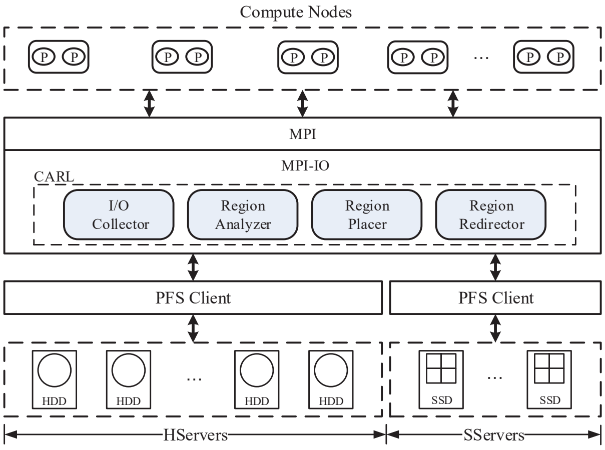


Fig. 2: Overview of system using S-CARL.

many data-intensive applications have predictable data access patterns [8], [44], thus I/O behavior can be obtained from previous runs. Based on this observation, the proposed static cost-aware region-level (S-CARL) data layout scheme divides a large file into several small regions, and then selectively places them on proper servers based on the region cost analysis.

4.1 System Overview

Fig. 2 shows the high performance cluster systems for which S-CARL is designed. In these systems, application processes on compute nodes access the data on file servers by calling the MPI-IO library. S-CARL resides in MPI-IO library and is responsible for placing data on the underlying HServers and SServers, which are accessed by a parallel file system respectively. S-CARL is independent of the file system; thus allowing the scheme to be portable and easily adopted to different file systems, such as PVFS [2], Lustre [3], and GPFS [4].

Fig. 3 shows the procedure of the static region-level data placement scheme, which consists of three phases. In the "Tracing Phase", the run-time statistics of I/O accesses are collected by *I/O Collector* during the applications' first execution. In the "Analysis Phase", *Region Analyzer* divides the file into regions and uses the data access cost model to estimate performance gains for file regions if they are placed on SServers over HServers. The performance gains are then used to generate a region gain table (*RGT*). In the "Placing Phase", *Region Placer* places file regions on the underlying servers according to *RGT*. In subsequent runs of the application, *Region Redirector* is added at the I/O middleware layer (MPI-IO library) to forward I/O requests to appropriate underlying HServers or SServers. Through these three phases, S-CARL reduces I/O time of the application in subsequent runs.

4.2 I/O Collector

I/O Collector is responsible for capturing run-time I/O access information of parallel applications. While there are other tools can be used, we use IOSIG [45] to obtain the information required by S-CARL. IOSIG is a pluggable library of MPI-IO, which utilizes the Profiling MPI interfaces (PMPI) to trace standard MPI-IO calls. After running applications with IOSIG, S-CARL can get the required information of file requests, such as process ID, MPI rank, file descriptor, type of operation, offset, request size, and time stamp.

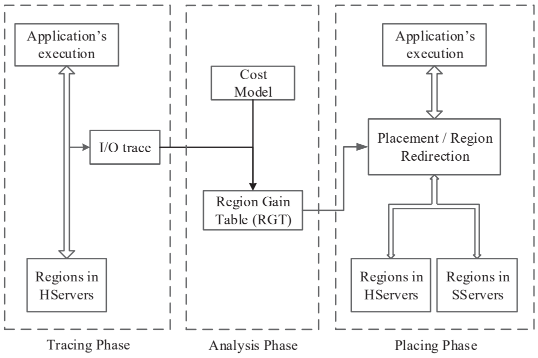


Fig. 3: The procedure of data placement scheme.

4.3 Region Analyzer

Region Analyzer evaluates the performance gain of placing a file region on SServers over HServers. The basic approach includes the following three steps.

First, the address space of the file is logically divided into regions by a fixed chunk size (e.g. 64MB or 128MB) for further analysis. The smaller the region size, the more efficient will be the data placement. However, operating at the region level incurs metadata overhead to keep track of region locations and other statistics and this overhead is inversely proportional to the region size. We choose a region size of 64MB with an acceptable system overhead.

Second, 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 spans across several regions, then each subpart of the request contributes to the region it belongs to.

Third, the performance gain of placing each region on SServers instead of HServers 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 taken by HServers and SServers to serve the jth request respectively, which are calculated using Equation (1) and (8), then the gain g_i for the ith file region is defined by

$$g_i = \sum_{j=1}^{n(i)} (T_H^j - T_S^j) \tag{11}$$

To make appropriate data placement decisions, the cost gains of all regions are stored in a global region gain table (*RGT*), which will be used by *Region Placer*. Since *RGT* comprehensively considers the key factors in data

Algorithm 1 The region-level data placement algorithm

```
Require: I/O Request: $r$, Region gain table: $RGT$, Region map table: $RMT$.
/* Lookup $r$ in $RMT$, return a mapping entry $reg$*/
$reg \leftarrow RMT\_lookup(r)$
if $reg$ != $NULL$ then
    if $reg.tier$ == $SServers$ then
        Forward $r$ to $reg$ on $SServers$
    else
        Forward $r$ to $reg$ on $HServers$
    end if
else
    /* Otherwise, place data to a new region */
    $c \leftarrow$ Calculate the free capacity of $SServers$
    Let $k = c/region\_size$
    /* Find top k regions in $RGT$ but not in $RMT$ */
    $Reg[k] \leftarrow$ top_k($\{x : x \in RGT \wedge x \notin RMT\}$)
    /* Find a matched region in $Reg[k]$ */
    for each $reg \in Reg[k]$ do
        if $r$ in $reg$ and $reg.gain > 0$ then
            $reg \leftarrow$ Allocate a region from $SServers$
            Forward $r$ to $reg$ on $SServers$
        end if
    end for
    if no matched region found in $Reg[k]$ then
        $reg \leftarrow$ Allocate a region from $HServers$
        Forward $r$ to $reg$ on $HServers$
    end if
    Add an entry of $reg$ into $RMT$
end if
```

accesses, such as number of requests, request frequency, request size, and I/O parallelism of underlying servers, it can effectively guide the data placement in a multi-tiered I/O system.

4.4 Region Placer

Region Placer carries out the actual region placement on underlying HServers or SServers based on three factors: (1) the available free space on SServers, indicating whether SServers can accommodate the current region, (2) the performance gain in *RGT* for current 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 gain than other regions if it is located on SServers.

Algorithm 1 shows the data placement procedure for an incoming I/O request. First, a global region map table *RMT*, which keeps the location mapping information between a logical file region and a target region on HServers or SServers, is initialized. *RMT* is empty at the beginning, and will be continuously updated as new regions are allocated to the file. Upon a file write request, the algorithm checks if the request falls into a region that has been allocated by consulting *RMT*. If yes, the request is forwarded to the allocated region. Otherwise, a new region on SServers or HServers will be allocated to hold the request and the address of the allocated region is stored in the corresponding entry of *RMT*. Suppose there are k free regions on SServers, and the incoming request r belongs to logical file 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 the top-k unallocated regions in the descending order of their performance gains. Otherwise, the algorithm will allocate a free region on HServers to place the requested data. As the data location of a file does not change during the application's run, we call it is a *static region-level data placement scheme*.

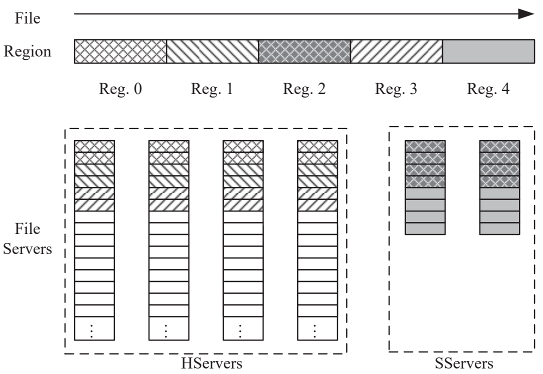


Fig. 4: An example of the static region-level data placement scheme.

Fig. 4 shows an example of the proposed data placement scheme. In this example, the file is divided into five regions, each having a different access cost. Among all 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 on SServers and the remaining regions are placed on 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 file requests with high performance. The region-level data placement scheme is a fine-grained optimization, and it is more suitable for applications with complicated data access patterns.

4.5 Region Redirector

Region redirector in the MPI-IO library is responsible for redirecting user's I/O requests to underlying HServers or SServers. Upon a file request, *Region Redirector* first determines the requested logical file regions based on the request offset, request size, and region size. Then it examines *RMT* with the logical file regions to find the target regions. Finally, the read/write operations will be forwarded to the target regions on underlying HServers or SServers. All the operations are transparent to applications. In this way, SServers, which have a small storage space, can be intelligently utilized according to the I/O patterns.

5 The Dynamic Region-Level Data Placement

In the previous section, we described a static region-level data placement in a multi-tiered parallel I/O system. While effective to optimize I/O performance, the static tiering technique is based on the complete knowledge of I/O workloads of applications. However, in practical systems, this assumption may be unrealistic, and data placement scheme needs to adapt dynamically to run-time changes of workloads.

5.1 Basic Idea of Dynamic Placement

To address this issue, we propose a dynamic region-level data placement scheme (D-CARL), which leverages data migration to improve parallel I/O performance at the runtime. The basic idea of D-CARL is to divide the entire workload duration into multiple time windows, and place file regions on proper servers based on the workload within each window. As workloads on certain file regions may change as time elapses, placing them on the old type of servers may offset the parallel I/O performance. In such cases, D-CARL will migrate them from the old locations to the new storage tiers accordingly. The efficiency of the data migration depends on the locality of the workloads. Fortunately, workloads of many applications show locality characteristics [17], [18]. Unlike S-CARL that does not change file region locations duration the entire observation window, such a dynamic placement scheme can adaptively accommodate varying I/O workloads.

5.2 Design of D-CARL

5.2.1 Time Window Scale

A first issue is the time scale at which file regions move across different types of servers. One choice is to place regions once during system instantiation or move them at coarse grain intervals of the order of hours or days. However, previous studies show that I/O workload changes typically most of the time [46], this semi-static placement is not the optimal. The alternate choice is to move region at intervals on the order of minutes or hours. Such a system exploits variations in region workload to improve its efficiency. Dynamic migration of the regions into the proper storage tier (HServers or SServers) when required enables cost-effective use of the resources. In this study, we choose to perform dynamic data placement with a time window of 10 minutes, which depends on the workloads and can ensure the migration overhead does not overwhelm its benefit. Previous study also uses a time window of the order of minutes [18].

5.2.2 Data Migration Algorithm

The second concern is how to determine which data need to migrate and where to migrate. As we discussed, since region-level data placement is beneficial to I/O performance, D-CARL migrates data at the *region granularity*. From the viewpoint of SServers, there are two types of regions that need to be migrated. The first are the ”outgoing” regions which must be moved to HServers; they are no longer beneficial enough to be on SServers. The second are the ”incoming” regions which now have a sufficiently high performance gain to be migrated to SServers but are not currently on them. These constitute the regions to be accessed to ensure high parallel I/O performance in the future.

Algorithm 2 The dynamic region migration algorithm

```
Require: Region gain table: $RGT$, Region map table: $RMT$.
$outgoinglist \leftarrow \emptyset$
$incominglist \leftarrow \emptyset$
/* Save top-$k$ ranked regions in the last time window */
$Reg\_las[k] \leftarrow$ top_k($\{x : x \in RGT\}$)
Update $RGT$ based on I/O accesses in the current time window
/* Find top-$k$ ranked regions in the current time window */
$Reg\_cur[k] \leftarrow$ top_k($\{x : x \in RGT\}$)
/* Construct the outgoing list */
for each $reg \in Reg\_las[k]$ do
    $tier \leftarrow RMT\_lookup\_tier(reg)$
    if $tier == SServers$ and $reg \notin Reg\_cur[k]$ then
        $outgoinglist \bigcup \{reg\}$
    end if
end for
/* Construct the incoming list */
for each $reg \in Reg\_cur[k]$ do
    $tier \leftarrow RMT\_lookup\_tier(reg)$
    if $tier! = SServers$ then
        $ingoinglist \bigcup \{reg\}$
    end if
end for
```

Algorithm 2 shows the data migration algorithm which is responsible for constructing the migration plan and scheduling the migration. It is executed at the end of each time window. First, the global region gain table *RGT* and region map table *RMT* are initialized. *RMT* is empty at the beginning and continuously updated as new regions are allocated for the file. Then, at the end of each time window, D-CARL updates *RGT* based on the recent data accesses in the last time window. Assume that SServers have *k* regions, the outgoing and incoming lists are then created based on the top-*k* entries in *RGT* and contents in *RMT*. The algorithm walks through each region on SServers by looking up *RMT*, creating an entry in the outgoing list for each region that is no longer in the top-*k* entries in *RGT*. Then, it walks through each entry in the top-*k* ranked regions, creating an entry in the incoming list for each region which is currently not on SServers. Once these two steps are completed, the new migration plan is obtained. The algorithm begins with no prior knowledge about the workload, then periodically gathers I/O access information, learns workload behavior and makes migration plan to appropriate locations in response to workload characteristics.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2636837, IEEE Transactions on Parallel and Distributed Systems

5.2.3 Data Migrator

The third issue is how to migrate data between HServers and SServers. The actual data migration operation is carried out by *Data Migrator* at the end of each time window. It includes two distinct phases: SServers to HServers and vice versa. These two phases are treated differently. The first phase, SServers to HServers, addresses operations in the outgoing list of the new placement plan. For each entry in the list, the data movement operation is followed by coping data from SServers to HServers and updating the corresponding entry in *RMT*. The second is HServers to SServers phase, which handles the incoming list in a similar way and updates the new entry in *RMT*.

5.3 Implementation

We implement D-CARL in MPI-IO library MPICH2 and parallel file system OrangeFS. The primary challenges are discussed below.

5.3.1 Key Data Structures

In the proposed placement scheme, *RGT* and *RMT* are two key mapping tables to store region performance gains and mapping relationship between logical file regions and target file regions. We use *Berkeley DB* [47] to implement *RGT* and *RMT*, each being a database file in a standalone space on SServers. The *Berkeley DB* is configured as a hash table, and each record is a key-value pair. For *RGT*, the key is the RegionID encoded with application name, number of processes, rank of the process, original file name, and region sequence; the value contains the performance gain. For *RMT*, the key is also RegionID but the value is the target region information. For a parallel application, there may be multiple processes accessing the two shared tables at the same time, which may lead to access contention. However, with the light-weighted database, the contention issue is addressed and metadata operations are performed efficiently. Additionally, we use a list to maintain most frequently accessed entries for each table in memory to speed up lookups.

5.3.2 I/O Redirection in MPI-IO

We modify the MPI library so that the mapping table *RMT* is loaded with `MPI_Init()` and unloaded with `MPI_Finalize()`. To keep track of the location of each original logical file region, *RMT* is stored in a file in the same directory of the MPI program. The mapping table entries are also hashed in memory for efficient table lookup. Changes made to the mapping entries in memory are synchronously written to the storage to survive power failures. 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 servers.

5.3.3 Data Migration Issues

To avoid interfering with the normal MPI I/O operations, D-CARL creates a new I/O helper thread in each process to handle the background data movement. This I/O thread is created when the process opens the first file by calling `MPI_File_open` and destroyed after the last file is closed with `MPI_File_close`. While each process can have multiple files opened, only one migration thread is created. Once the I/O thread is created, it enters an infinite loop to perform data migration operations until it is signaled for termination. It communicates with the main thread through shared variables that store file access information, such as file handler, offset, etc.

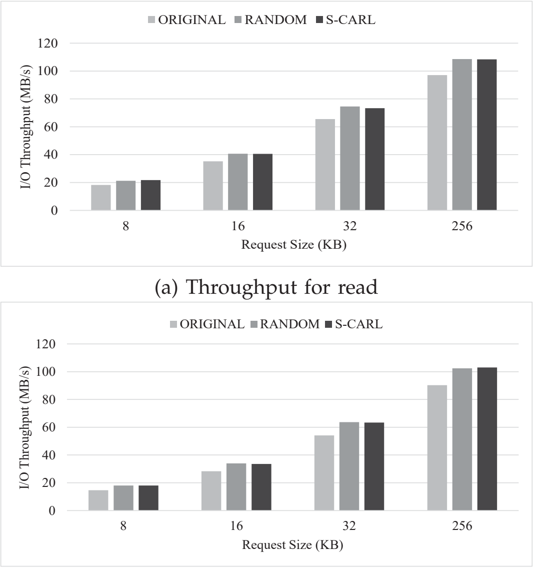
6 Performance Evaluation

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

6.1 Experimental Setup

We conduct the experiments on a 65-node SUN Fire Linux cluster, where each node has two AMD Opteron(tm) processors, 8GB memory, and a 250GB HDD. 16 nodes are equipped with additional OCZ-REVODRIVE 100GB SSD. All nodes are equipped with Gigabit Ethernet interconnection. The operating system is Ubuntu 13.04, the MPI-IO library is MPICH2-1.4.1p1, and the parallel file system is OrangeFS 2.8.6. Among the available nodes, we select eight as computing nodes, eight as HServers, and four as SServers. Both HServers and SServers are accessed through an OrangeFS parallel file system respectively. By default, data is striped over file servers with a 64KB striping unit size. In general, the larger the capacity of an SServer, the better the I/O performance. To avoid the overestimation of performance improvement, we set the data size on SServers as 20% of the application file size.

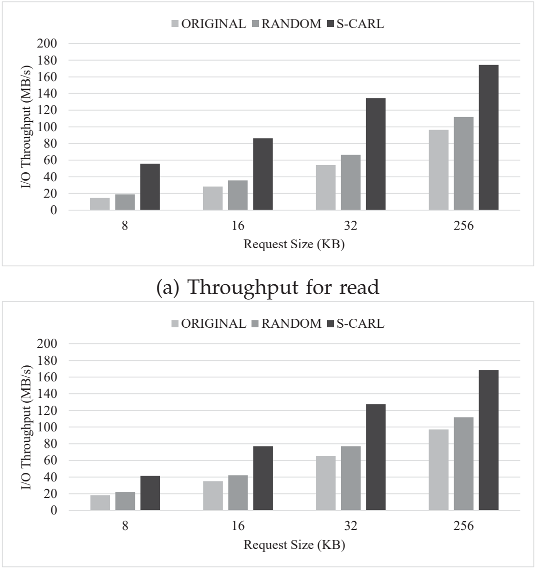
We use the popular benchmark IOR [48], HPIO [49], and a real application [50] to evaluate the proposed data placement scheme. First, we show the efficiency of the static region-level data placement policy (S-CARL) when the application's workload is known. We compare S-CARL with two other static counterparts: RANDOM and ORIGINAL. RANDOM distributes file regions on underlying HServers or SServers randomly. ORIGINAL places file regions only on HServers, which results in the worst-case system performance. Second, we evaluate the efficiency of the dynamic region-level data placement policy (D-CARL) by comparing it with RANDOM and S-CARL when we have no knowledge about the workloads.



(a) Throughput for read

(b) Throughput for write

Fig. 5: I/O throughputs with varying request sizes under the uniform random workload.



(a) Throughput for read

(b) Throughput for write

Fig. 6: I/O throughputs with varying request sizes under the Zipfian random workload.

6.2 Evaluation on Static Region-Level Data Placement

6.2.1 IOR Benchmark

To simulate different I/O patterns, we use IOR to generate two kinds of workloads. One workload generates uniform random requests by using the default implementation of IOR. The other generates a Zipfian distribution by modifying the default implementation of IOR. With such access patterns, one can see the impact of skewness in workload on the performance behavior of S-CARL.

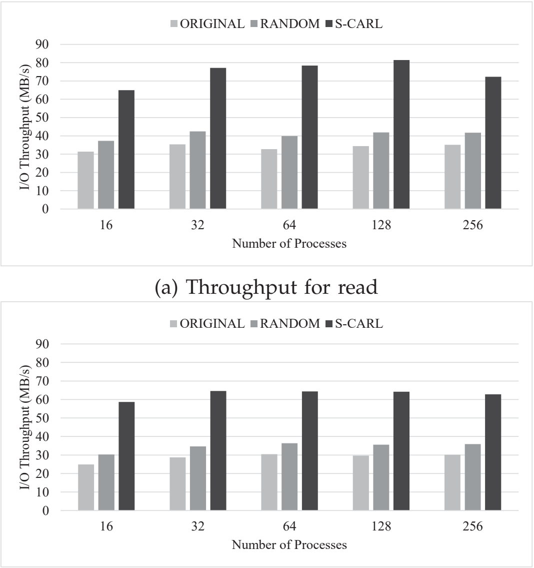
Varying Request Sizes: We run IOR with request sizes of 8KB, 16KB, 32KB, and 256KB, respectively. The number of processes is fixed to 32. Each process is responsible for accessing its own part of a 10GB shared file, and continuously issues requests with random offsets. Fig. 5 shows the I/O performance under the uniform workload with various data placement schemes. For read requests, both S-CARL and RANDOM can improve the original I/O throughput by adding SSDs to the parallel I/O system. S-CARL improves read performance of ORIGINAL by 18.8%, 15.3%, 12.1% and 11.7%, respectively, in terms of different request sizes. With a larger request size, the I/O throughput improves because serving larger requests on SServers leads to higher I/O bandwidth. Compared with RANDOM, S-CARL has a similar performance behavior. This is because with a uniform workload the regions selected by S-CARL nearly bring the same performance gain as RANDOM. Under this case, S-CARL nearly degrades to RANDOM. The write test shows similar results.

Fig. 6 shows the I/O performance comparison under the Zipfian workload. S-CARL can improve read 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 S-CARL and RANDOM provide better performance compared to the results with a uniform workload. However, S-CARL has a significant performance improvement over RANDOM. This is because S-CARL places the most frequently accessed data on SSDs while RANDOM randomly selects data; thus, S-CARL can obtain more performance benefits. The write test yields similar results. In the comparison with ORIGINAL, S-CARL increases the throughput by 127.4%, 118.9%, 94.8% and 73.7%, respectively. Compared to RANDOM, S-CARL shows 88.1%, 82.3%, 65.6% and 51.0% improvements. However, S-CARL provides a more modest improvement in writes. This is because SSDs favor reads over writes.

Since S-CARL is unable to make significant performance improvement for the uniform random workload, we only focus on the workload with Zipfian distribution.

Varying Process Numbers: To show how the number of process affects I/O performance, we run IOR with 16 to 256 processes and the request size is fixed to 16KB. Fig. 7a shows the results of read performance with respect to Zipfian workloads. Similar to the previous test, S-CARL improves the overall bandwidth from 107.2% to 139.7%. As the number of processes increases, the I/O bandwidth first increases and then decreases, this is due to the fact that each HServer needs to serve requests from more processes and the competition among processes impedes the whole I/O progress. Fig. 7a shows another improvement of S-CARL: when the number of processes increases, the performance gain of S-CARL increases as well. In other words, S-CARL has better scalability

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2636837, IEEE Transactions on Parallel and Distributed Systems



(a) Throughput for read

(b) Throughput for write

Fig. 7: I/O throughputs with varying number of processes under the Zipfian random workload.

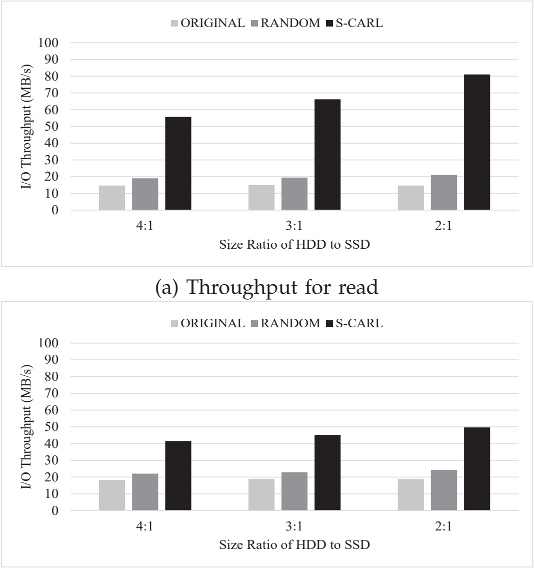
and hence are able to handle more concurrent I/O processes. Additionally, this figure shows that S-CARL is more effective than RANDOM, and shows performance improvements of 74.2%, 81.9%, 96.7%, 94.5% and 73.2% respectively. The performance trend is similar for write requests, as shown in Fig. 7b.

Varying SSD Sizes Generally the capacity of SServers is smaller than that of HServers and could be smaller than the I/O working set size of the application. To show the impacts of SSD space on the I/O performance, we run IOR by varying data size ratios of HServers to SServers from 4:1 to 2:1.

Fig. 8 shows the I/O throughputs for read and write operations. Similar to previous results, S-CARL outperforms RANDOM and ORIGINAL. S-CARL has performance improvements up to 278.7%, 356.4% and 450.8%, respectively, over the original I/O system performance in terms of different sizes of SServers. With the increased size of SServers, the I/O bandwidth improves because more high-cost data regions can be placed on and benefit from SServers. However, increasing the size of SServers will not substantially improve performance when high-cost regions are already stored on SServers.

6.2.2 HPIO Benchmark

HPIO is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate parallel I/O system performance [49]. This benchmark can generate various data access patterns by changing three parameters: region count, region spacing, and region size, which indicate the number of requests, the distance between two requests, and the request size, respectively. In our experiment, the region count is 4096, the region size is 128KB, and the region spacing is 32KB.



(a) Throughput for read

(b) Throughput for write

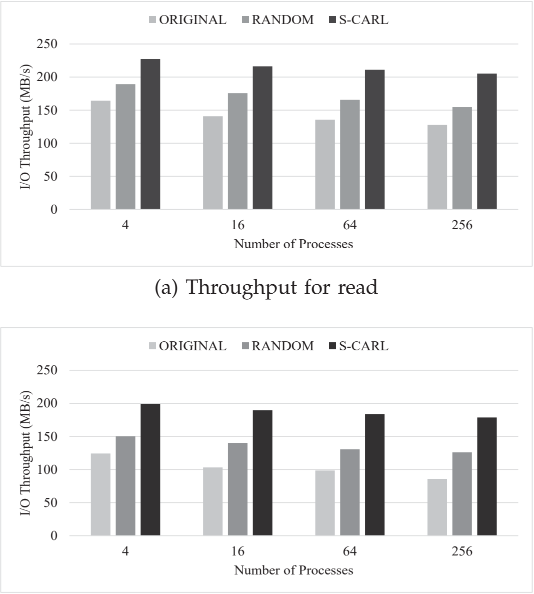
Fig. 8: I/O throughputs with varying SSD sizes under the Zipfian random workload.

We vary the number of processes to emulate access patterns with different I/O concurrencies. As shown in Fig. 9, S-CARL can increase both read and write throughput over ORIGINAL and RANDOM. For reads, S-CARL outperforms RANDOM by 20.1%, 23.3%, 27.4%, and 32.7% for 4, 16, 64, and 256 processes, respectively. As the number of processes increases, the performance speedup becomes more obvious because SServers have higher and more stable performance than HServers when serving a large number of processes. This confirms the adaptability of S-CARL: when the application's I/O accesses have a poorer throughput due to a higher I/O concurrency (more processes), more benefit is gained by using S-CARL. For write operations, the performance shows a similar trend as presented in Fig. 9b.

6.2.3 Real Application

Finally, we evaluate the performance of S-CARL with a real I/O trace from 'Anonymous LANL App 2' [50]. This application has a complex access pattern: each process of the application sends I/O requests with varied sizes at different parts of a shared file. Therefore, I/O workloads on different regions of the file show distinctive access characteristics. We replay the data accesses of the application according to the I/O trace to simulate the same data access scenario.

Fig. 10 shows the I/O throughput result. S-CARL obtains 108.5% and 69.3% performance improvement compared to ORIGINAL and RANDOM respectively. Although S-CARL can improve performance, the improvement is not as substantial as that of IOR under the Zipfian workload. This is because only a small part of the requests have different sizes while most of the requests in Zipfian workload have various access



(a) Throughput for read

(b) Throughput for write

Fig. 9: Throughputs of HPIO with varying numbers of processes.

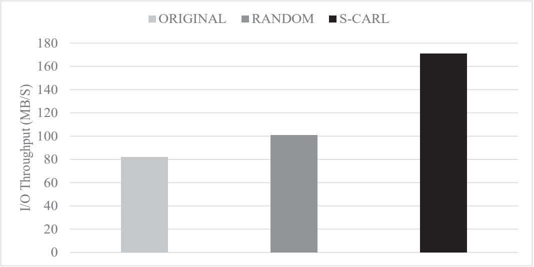


Fig. 10: I/O throughputs of LANL App2.

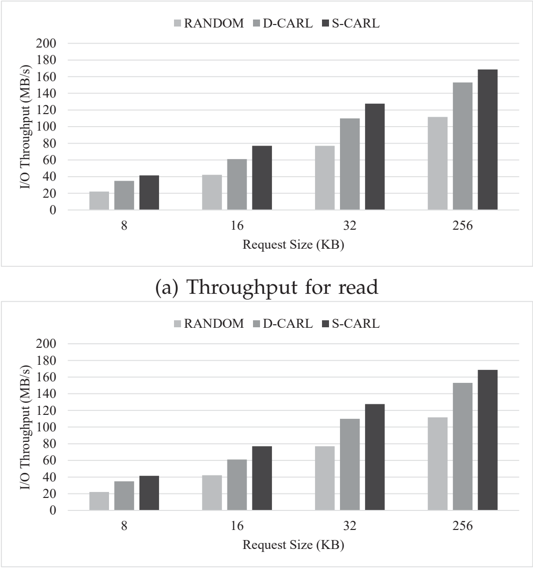
frequencies, which means that the region costs are not as skewed as those of Zipfian workload. However, the results indicate that S-CARL is an effective performance optimization method for applications with complex I/O access patterns.

6.3 Evaluation on Dynamic Region-Level Data Placement

We conduct experiments to show that the dynamic region-level data placement policy can improve I/O system performance, which verifies the need of data migration when the I/O workload changes and is unknown in advance. We compare D-CARL with RANDOM and S-CARL and omit ORIGINAL since ORIGINAL is the worst case.

6.3.1 The IOR Benchmark

We ran IOR with the Zipfian workload since it shows strong temporal locality and benefits the dynamic data migration.



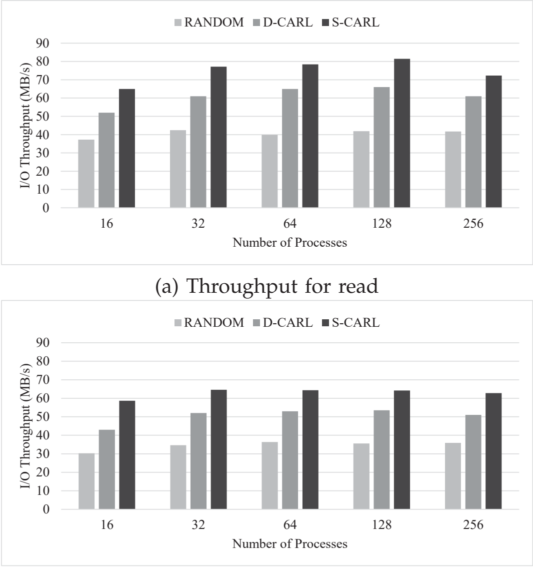
(a) Throughput for read

(b) Throughput for write

Fig. 11: I/O throughputs with varying request sizes.

We first run IOR with requests in different sizes of 8KB, 64KB, 512KB, and 4MB. As usual, the process number is fixed to 32. The corresponding I/O throughputs are shown in Fig. 11. Compared with RANDOM, D-CARL obtains 37.1%-157.5% additional performance improvements with respect to the different request sizes. We can see that S-CARL is the ideal case since it assumes the access patterns are known in advance. However, while D-CARL is not as good as S-CARL, the performance gap between them is not large.

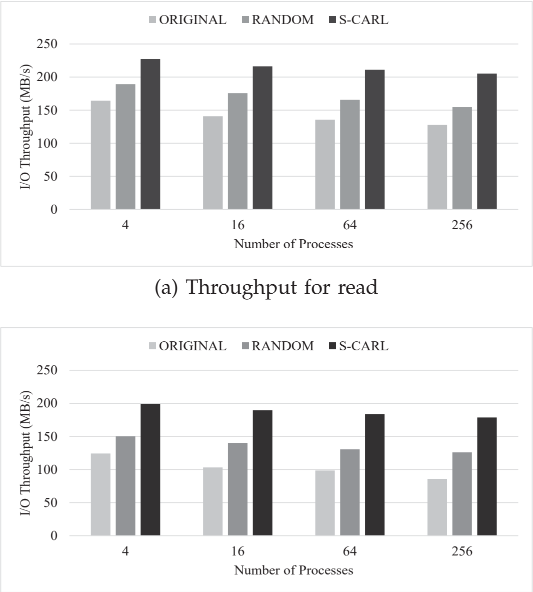
We also vary the number of processes. We run IOR with 8, 32, and 128 processes, and set the request size as 512KB. Fig. 12 describes the results of read and



(a) Throughput for read

(b) Throughput for write

Fig. 12: I/O throughputs with varying numbers of processes.



(a) Throughput for read

(b) Throughput for write

Fig. 13: Throughputs of HPIO with various numbers of processes.

write requests. Similar to the previous test, D-CARL is better than RANDOM but worse than S-CARL. D-CARL obtains 39.4%-63.2% performance improvements over RANDOM. Compared with S-CARL, D-CARL only suffers the performance degradation at most 15.6 %. This result shows that D-CARL is very effective to improve I/O performance even the workload is unknown in advance.

6.3.2 The HPIO Benchmark

We set the region count to 4096, the region size to 16KB, and the region spacing is 32kB. We vary the number of processes from 16 to 256. Fig. 13 shows the results. Similar to the IOR tests, D-CARL shows better performance than RANDOM but poorer performance than S-CARL. As the workload does not exhibit strong locality, the improvements obtained by D-CARL are not very significant. However, D-CARL still can outweigh RANDOM by 4.1%, 4.3%, 3.3%, and 4.8% respectively. Compared to the ideal case of S-CARL, D-CARL is closed to it and shows moderate performance.

6.3.3 Real Application

Finally, we evaluate the performance of D-CARL with the real application ’Anonymous LANL App 2’. As shown in Fig. 14, we find that D-CARL obtains 10.9% performance improvement compared to RANDOM. The improvements are not as significant as those of IOR. This is because the workload exhibits weaker locality than IOR so that the data migration policy brings less performance benefits.

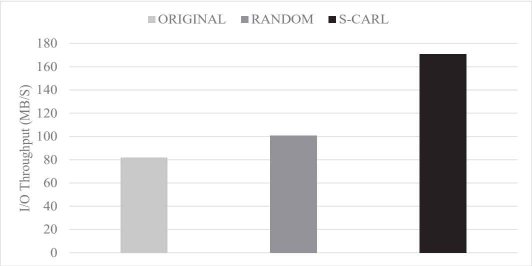


Fig. 14: Performance of LANL App2 under D-CARL.

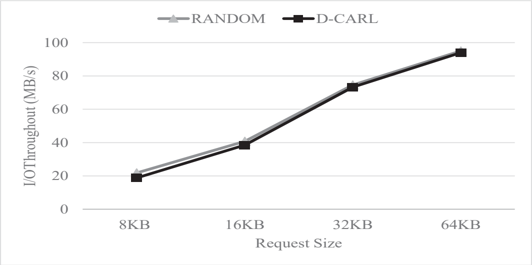


Fig. 15: Performance overhead.

6.4 Overhead Analysis

While the gains due to data placement are promising, CARL could incur some potential overhead on resource utilization.

6.4.1 Metadata Space Overhead

As described, to maintain data consistency, the region map table (*RMT*) is used in the I/O middleware to track data location on underlying servers. Furthermore, a region gain table *RGT* is used to determine the proper data placement. These two key tables would incur additional space overhead. In our implementation, the region size is fixed as 64MB. So for a 100GB PFS file, there are up to 1600 region entries in total. Since each entry in *RMT* is of several bytes, we assume 128 bytes for each, so the total size of *RMT* would be (1600 * 256) bytes, which is 0.4MB. Thus, the metadata space overhead is 0.4MB/100GB, which is less than 0.001% and even negligible for data sized in TB.

6.4.2 Performance Overhead

In S-CARL, *I/O Collector* uses IOSIG to collect trace files during the application’s first run. Previous work shows the overhead of IOSIG is very low [45] and this observation is also applied to our case. Since in our design the pattern analysis and planning are carried out only once in off-line fashion, the CPU and memory overhead is also acceptable for most HPC computing systems.

In D-CARL, some additional modules, such as those for collecting access information and constructing migration plan, could also incur performance overhead. To evaluate it, we run IOR with random workloads, so that the system would run additional modules without making actual data migration. The process number is fixed to 32, and the request size is varied from 8KB to 64KB. Fig. 15 demonstrates the introduced overhead is negligible.

7 Conclusions

Parallel I/O systems are widely used to mask the huge gap between CPU performance and disk drive performance. However, they may exhibit poor performance for certain I/O patterns. Newer solid state 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 approach to address the I/O performance issue.

In this paper, we propose 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. CARL includes a static policy S-CARL and a dynamic policy D-CARL. For applications whose I/O access patterns are completely known, S-CARL calculates the region costs within the entire workload duration, and uses a static data placement scheme to selectively place regions on the appropriate servers. To adapt to applications whose access patterns are unknown in advance, D-CARL uses a dynamic data placement scheme which migrates data among different servers within each time window.

Acknowledgment

This work was supported in part by the National Science Foundation of China under Grant No. 61572377, No. 61672513, U1401258 and 61550110250, China National Basic Research Program (973 Program, No. 2015CB352400), the Natural Science Foundation of Hubei Province of China under Grant No. 2014CFB239, and the Open Fund from HPCL under Grant No. 201512-02, the Open Fund from SKLSE under Grant No. 2015-A-06, and the Science and Technology Planning Project of Guangdong Province (2015B010129011, 2016A030313183). Yang Wang is the corresponding author.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[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 on Cluster, Cloud and Grid Computing (CCGrid)*, 2011, pp. 414–423.

[8] 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, pp. 345–356.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S. He, X.-H. Sun, B. Feng, X. Huang, and K. Feng, "A Cost-Aware Region-Level Data Placement Scheme for Hybrid Parallel I/O Systems," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2013, pp. 1 – 8.

[16] H. Wang and P. Varman, "Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, 2014, pp. 229–242.

[17] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, 2014, pp. 33–45.

[18] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost Effective Storage Using Extent Based Dynamic Tiering," in *Proccedings of the 9th conference on File and Storage Technologies*, 2011, pp. 273 – 286.

[19] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," in *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, 23-27 May 2011 2011, pp. 1–14.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Y. Wang and D. Kaeli, "Profile-Guided I/O Partitioning," in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003, pp. 252–260.

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

[27] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "Borg: Block-Reorganization for Self-

Optimizing Storage Systems," in *Proccedings of the 7th conference on File and Storage Technologies*, San Francisco, California, 2009, pp. 183–196.

[28] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang, "EDM: An Endurance-Aware Data Migration Scheme for Load Balancing in SSD Storage Clusters," in *Proceedings of 28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

[29] 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.

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

[31] H. Song, H. Jin, J. He, X.-H. Sun, and R. Thakur, "A Server-Level Adaptive Data Layout Strategy for Parallel File Systems," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, 2012, pp. 2095–2103.

[32] Z. Gong, D. A. B. II, X. Zou, Q. Liu, N. Podhorszki, S. Klasky, X. Ma, and N. F. Samatova, "PARLO: PArallel Run-time Layout Optimization for Scientific Data Explorations with Heterogeneous Access Patterns," in *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013.

[33] W. Tantisiriroj, S. Patil, G. Gibson, S. Seung Woo, S. J. Lang, and R. B. Ross, "On the duality of data-intensive file system design: Reconciling hdfs and pvfs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[34] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[35] 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.

[36] X. Zhang, K. Liu, K. Davis, and S. Jiang, "iBridge: Improving Unaligned Parallel File Access with Solid-State Drives," in *Proceedings of 27th IEEE International Parallel and Distributed Processing Symposium*, 2013, pp. 381–392.

[37] S. He, X.-H. Sun, B. Feng, and F. Kun, "Performance-aware data placement in hybrid parallel file systems," in *Proceedings of the 14th International Conference on Algorithms and Architectures for Parallel Processing*, 2014, pp. 563–576.

[38] S. He, Y. Liu, and X.-H. Sun, "A Performance and Space-Aware Data Layout Scheme for Hybrid Parallel File Systems," in *Proceedings of the Data Intensive Scalable Computing Systems Workshop*, 2014, pp. 41–48.

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

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

[41] S. He, Y. Wang, and X.-H. Sun, "Boosting Parallel File System Performance via Heterogeneity-Aware Selective Data Layout," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2492–2505, 2016.

[42] S. He, X.-H. Sun, and Y. Wang, "Improving Performance of Parallel I/O Systems through Selective and Layout-Aware SSD Cache," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 10, pp. 2940–2952, 2016.

[43] X. Zhang, S. Jiang, and K. Davis, "Making Resonance a Common Case: A high-performance Implementation of Collective I/O on Parallel File Systems," in *Proceedings of 23th IEEE International Parallel and Distributed Processing Symposium*, 2009, pp. 1–12.

[44] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014, pp. 213–228.

[45] 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.

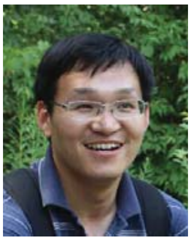
[46] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads," in *USENIX Annual Technical Conference*, 2008.

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

[48] "Interleaved Or Random (IOR) Benchmarks," 2016. [Online]. Available: http://sourceforge.net/projects/ior-sio/

[49] A. Ching, A. Choudhary, W.-k. Liao, L. Ward, and N. Pundit, "Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

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



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

Yang Wang received the BSc degree in applied mathematics from Ocean University of China (1989), and the MSc and Ph.D. degrees in computer science from Carleton University (2001) and University of Alberta, Canada (2008), respectively. He is currently with Shenzhen Institute of Advanced Technology, Chinese Academy of Science, as a professor. His research interests include cloud computing, big data analytics, and Java Virtual Machine on multicores.

Zheng Li received his Ph.D. degree in Computer Science from Illinois Institute of Technology, in USA. He is currently an assistant professor in the School of Computer Science at Western Illinois University. His research interests include distributed computing, real-time computing, many-core computing, and reconfigurable computing.

Xian-He Sun received his BS in Mathematics in 1982 from Beijing Normal University, China, and completed his MS and Ph.D. degrees in Computer Science in 1987 and 1990 from Michigan State University. He is a distinguished professor of the Department of Computer Science, the Illinois Institute of Technology (IIT), Chicago. He is the director of the Scalable Computing Software laboratory at IIT, and is a guest faculty in the Mathematics and Computer Science Division at the Argonne National Laboratory. His research interests include parallel and distributed processing, memory and I/O systems, software systems, and performance evaluation and optimization. He is a fellow of the IEEE.

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