Improving Performance of Parallel I/O Systems through Selective and Layout-Aware SSD Cache

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

Abstract—Parallel file systems (PFS) are widely-used to ease the I/O bottleneck of modern high-performance computing systems. However, PFSs do not work well for small requests, especially small random requests. Newer Solid State Drives (SSD) have excellent performance on small random data accesses, but also incur a high monetary cost. In this study, we propose SLA-Cache, a Selective and Layout-Aware Cache system that employs a small set of SSD-based file servers as a cache of conventional HDD-based file servers. SLA-Cache uses a novel scheme to identify performance-critical data, and conducts a selective cache admission (SCA) policy to fully utilize SSD-based file servers. Moreover, since data layout of the cache system can also largely influence its access performance, SLA-Cache applies a layout-aware cache placement scheme (LCP) to store data on SSD-based file servers. By storing data with an optimal layout requiring the lowest access cost among three typical layout candidates, LCP can further improve system performance. We have implemented SLA-Cache under the MPICH2 I/O library. Experimental results show that SLA-Cache can significantly improve I/O throughput, and is a promising approach for parallel applications.

Index Terms—Parallel I/O system, I/O middleware, solid state drive, cache system

1 Introduction

Data access has become the major performance bottleneck of modern computer systems. Over the past three decades, processor speeds have increased nearly 50 percent per year, but disk speeds have only improved by roughly 7 percent [1]. Despite the large performance disparity between processor and storage device, many scientific applications are becoming increasingly data intensive in the high-performance computing (HPC) domain. For example, the *astro* program in astronomy, generates tens of gigabytes of data in one run [2]. Such enormous data requirements are putting unprecedented pressure on HPC systems.

To meet the ever increasing I/O demands, HPC clusters rely on parallel I/O systems to provide efficient data services. Parallel I/O system consists of several layers including application, I/O middleware, parallel file system (PFS), and storage system layer. In general, a parallel file system, such as PVFS [3], Lustre [4], and GPFS [5], will stripe file data across multiple file servers. By allowing file requests to be concurrently served by multiple nodes, the I/O system performance can be significantly improved.

While PFSs are an effective approach to increase the I/O performance of large requests, they fail to perform well for small requests, especially random requests. One reason is that small requests lead to poor I/O parallelism among multiple servers. The other is that hard disk drives (HDD), the dominant storage media on current servers, are notoriously slow in random data access due to the mechanical nature of disk head movements. Small random requests have become the number one performance killer of PFSs [6], [7], [8].

A number of approaches have been proposed in the I/O hierarchy to speed up the parallel I/O system performance. I/O middleware techniques increase disk throughput by transforming a large number of small and non-contiguous requests into large contiguous requests [9], [10]. Memory caching strategies reduce the I/O latency by accessing more data from high-speed memory instead of storage devices [11], [12]. I/O scheduling approaches reorganize the incoming I/O requests to create more sequential accesses in order to improve performance [13]. These methods are very helpful, however, they need to be extended to take the advantage of the availability of new technologies, such as solid state drives (SSDs).

Advanced storage devices, such as SSDs, provide a promising solution to improve small random accesses. SSDs can provide low I/O latency, high data bandwidth and low power consumption, thus are attracting attention in HPC systems [14]. Generally, an SSD is commonly used as a cache of HDD [6], [7], [15] or as a persistent storage device on a single file server [16], [17]. While straightforward to implement, these approaches require a large number of SSDs in a large-scale storage cluster, thus may be costly and inflexible. Furthermore, since SSDs are deployed on each file server, the global utilization of SSDs becomes impossible though it can be very useful to improve I/O system performance [18], [19].

In this paper, we propose SLA-Cache, a Selective and Layout-Aware SSD Cache architecture to combine the merits of SSDs with a parallel file system. The main idea is to

- *S. He is with the State Key Laboratory of Software Engineering, Computer School, Wuhan University, Luojiashan, Wuhan 430072, Hubei, China, and the State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha 410073, Hunan, China. E-mail: heshuibing@whu.edu.cn.*
- *Y. Wang is with the Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Xueyuan Avenue 1068, Shenzhen University Town, Shenzhen 518055, China. E-mail: yang.wang1@siat.ac.cn.*
- *X.-He Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: sun@iit.edu.*

Manuscript received 31 July 2015; revised 16 Jan. 2016; accepted 18 Jan. 2016. Date of publication 25 Jan. 2016; date of current version 14 Sept. 2016.
Recommended for acceptance by H. Jin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2521363

employ a small set of SSD-based file servers (SServers) as a cache of conventional HDD-based file servers (HServers), and apply two polices to improve the cache system performance. To make full utilization of the limited space of SServers, SLA-Cache uses a selective cache admission (SCA) policy to only buffer or cache large amounts of performance-critical data for both read or write requests. To this end, SLA-Cache develops a cost model in a parallel I/O system to identify the performance-critical data. Furthermore, since data layout of a parallel file system can largely affect the I/O performance of the cache system, SLA-Cache applies a layout-aware cache placement (LCP) policy to stores data on SServers. Instead of using a fixed data layout, SLA-Cache places data on SServers with an optimal layout requiring the lowest access cost among three typical layouts. Compared to the cache system adopting a layout-oblivious placement scheme in our conference version [8], SLA-Cache can further improve I/O system performance.

Conventionally, a cache system uses data locality principals to increase cache efficiency. However, SLA-Cache is designed to utilize an SSD's ability to support small random data accesses. Therefore, the selection algorithm of SLA-Cache is derived from the available performance benefits of SSD-based file servers, not the data access locality. Application-aware scheduling to utilize data access performance on SSDs and the parallelism of PFS is one key strength of SLA-Cache. Moreover, SLA-Cache stores data on SServers in a layout-aware style, which is the first effort to further improve cache system performance in a parallel I/O system.

In summary, we make the following contributions.

- We introduce a cost model for parallel file systems, which evaluates the access time of a file request on HServers under one data layout, and on SServers under three typical data layouts.
- We propose a selective and layout-aware caching scheme, which first identifies performance-critical data by analyzing the data access cost under different layouts, and then uses a selective cache admission and layout-aware cache placement policy to take full advantage of the hybrid SSD and PFS architecture.
- We have implemented SLA-Cache under the MPICH2 I/O library. Experimental results with representative benchmarks show that SLA-Cache can significantly improve the I/O performance of an original I/O system, a cache system with non-selective caching admission policy, as well as a cache system with selective but layout-oblivious caching admission policy.

The rest of this paper is organized as follows. The background and motivation are given in Section 2. Section 3 describes the design of SLA-Cache and Section 4 presents the detailed implementation. Section 5 evaluates the performance of SLA-Cache with representative benchmarks. Section 6 discusses the related work. Finally, we conclude the paper in Section 7.

2 Background and Motivation

2.1 PFS Performs Poorly for Small Random Requests

Due to the nature of HDDs and the low parallelism of servers, PFSs usually show poor performance for small random requests. To illustrate this, we ran IOR [20] on a PVFS2 file system with eight file servers (each includes a single HDD). We set the number of processes to 16, the overall file size to 16 GB, and vary the request size from 4 KB to 32 MB. Each of the n MPI processes reads its own $1/n$ of the shared file, and continuously issues requests with random offsets.

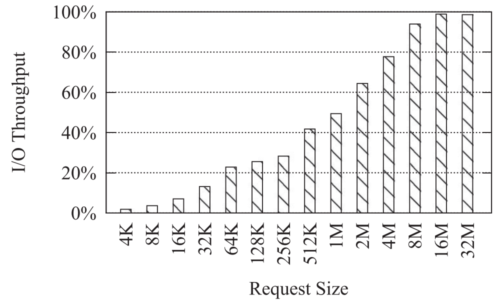


Fig. 1. I/O throughput normalized to the maximal system performance for random reads with different request sizes.

Fig. 1 demonstrates the aggregated throughput for different request sizes during random I/O operations. The average throughput is under 30 percent of the maximal system throughput with different request sizes from 4 to 256 KB. For request size larger than 8 MB, the I/O performance shows small variance and is approaching the maximal I/O system performance. These results confirm that small random access is a major performance impediment to parallel I/O systems.

2.2 SSD Has Workload-Dependent Performance Advantages

As SSDs are completely built on semiconductor chips, they provide much higher data transfer rate and lower access latency than HDDs. To show the performance advantages of SSDs, we tested I/O throughput of an OCZ-RevoDrive X2 SSD and a Seagate HDD in our experiments. We make two important findings for different access patterns. First, SSD has the most significant performance gain in random data accesses with small request sizes, for both reads and writes. For example, with a request size of 4 KB, SSD achieves more than 14.5 times and 27.2 times higher throughputs than HDD for random reads and writes, respectively. Second, the relative performance gains of reads and writes diminish as request size increases. For example, if the request size increases to 256 KB, the relative performance gains of sequential reads and writes diminish to 2.2 times and 1.9 times. Similar findings can be found in [16].

These observations indicate that performance benefits of a single SSD are highly dependent on workload access patterns. In terms of a parallel I/O system, the achievable performance benefits of the SServers are also dependent on the number of the HServers and the number of the SServers, because the I/O parallelism is very important to the aggregate I/O performance. We must identify the data that can bring the most performance benefits and migrate them into the SServers. SLA-Cache identifies the performance critical data via a proposed cost model, and then uses a smart selective cache admission policy to make full utilization of the limited SServer space.

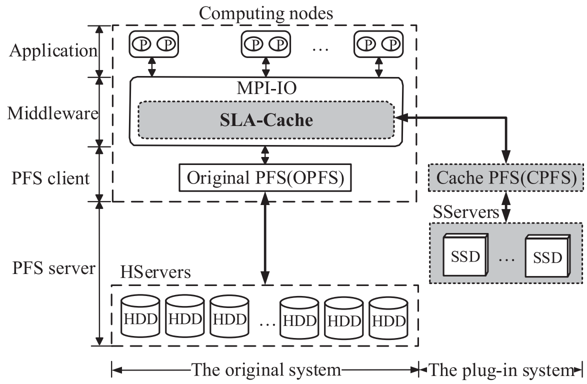


Fig. 2. The SLA-Cache architecture overview.

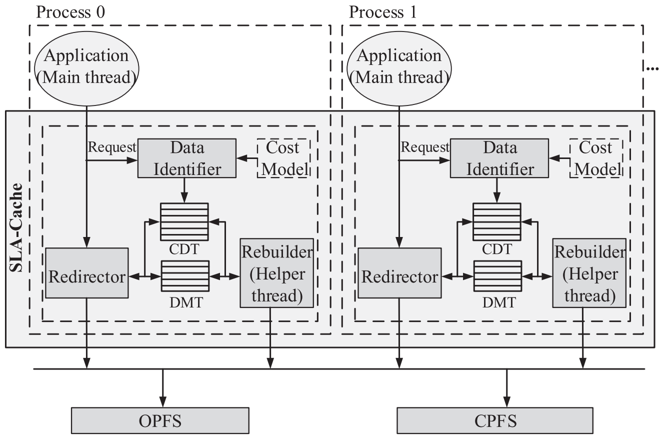


Fig. 3. The software structure of SLA-Cache.

2.3 Data Layout Affects the Cache Performance

A parallel file system, such as PVFS [3], Lustre [4], and GPFS [5], supports three typical data layout policies—one-dimensional horizontal (*1-DH*), one-dimensional vertical (*1-DV*), and two-dimensional (*2-D*) data layout [21]. *1-DH* is the simple striping method that distributes a process's data across all available servers; *1-DV* performs no striping at all, and instead places the file data on one server; *2-D* is a hybrid method, it distributes the file on a subset of servers.

One can choose to distribute file data on servers with a desired layout policy. As described in our prior work [18], [21], each layout policy is only suitable for a specific kind of I/O patterns and server configurations. For example, when the number of processes is far less than the number of servers and the request size is very large, *1-DH* brings the best I/O performance among the three policies. However, when the number of processes is much larger than the number of servers, *1-DV* would be the best choice, because each server has to serve requests from all processes, incurring severe I/O contention which can significantly degrade the system performance.

Since data layout in a parallel file system can largely affect I/O performance, data should be stored on the underlying servers with a proper layout. In SLA-Cache, we do not change the original data layout (*1-DH* by default) on HServers, but we store the data on SServers in a layout-aware style. With an optimal layout requiring lowest access cost among the three typical layout candidates, the cache system performance can be further improved.

3 SLA-Cache Design

SLA-Cache aims to use SSD-based file servers to cache performance-critical data of a parallel I/O system. By exploiting performance advantages of SSD-based file servers for small random requests, SLA-Cache can significantly improve the I/O system performance.

3.1 Architecture Overview

Fig. 2 shows the high performance computer systems for which SLA-Cache is designed. SLA-Cache acts as an augmented module to MPI-IO library [22], which is a middleware between applications and underlying PFSs. In these systems, besides the traditional HDD-based file servers, there are a small number of SSD-based file servers. HServers are accessed by the original parallel file system (OPFS); SServers act as a fractional cache of HServers and are accessed by the cache parallel file system (CPFS). When application processes pass their I/O requests to MPI-IO, SLA-Cache intercepts all the requests and chooses the proper servers to serve them.

Positioning SLA-Cache at the middleware layer is ideal for several reasons. First, key global data access information can be used to improve performance. Second, the middleware layer is independent of the file system, allowing the solution to support multiple file systems, such as PVFS [3], Lustre [4], and GPFS [5]. Third, the plug-in design is transparent to applications, therefore user programs do not require any modifications. Finally, because only a small cluster of SSDs are deployed into the system, the design is flexible and highly cost-effective.

Fig. 3 presents the key software modules of SLA-Cache. It includes three components: *Data Identifier*, *Redirector* and *Rebuilder*. *Data Identifier* intercepts every file request issued to HServers, and identifies requests for performance-critical data using a data access cost model. *Redirector* redirects the selected requests to the high-performance SServers. While selected write requests and cached read requests are redirected to SServers, other write requests and missed read requests are directed to the traditional HServers. *Rebuilder* is responsible for flushing the selected write data back to HServers, and fetching selected read data to SServers. In addition to the functional components, SLA-Cache maintains two important data tables, CDT and DMT, to recognize performance-critical data and keep track of cached data.

3.2 Data Access Cost Model

Since SServer has a relatively small storage capacity, it is cost-efficient for SLA-Cache to only cache performance-critical data. Thus, the potential performance benefit of redirecting a request to SServers must be evaluated to prioritize their eligibility for caching. To this end, a cost model is derived to evaluate the data access time for each file request in a parallel file system.

We consider three data layouts: *1-DH*, *1-DV*, and *2-D*, as shown in Fig. 4. We choose these layouts because they are typical and widely used in current parallel file systems. Since we do not change the data layout on HServers, we assume the original data are placed on HServers with the default *1-DH* layout. But for the cache data, we assume them can be distributed on SServers with one of the three layouts to achieve best performance. Table 1 lists the corresponding parameters in the model.

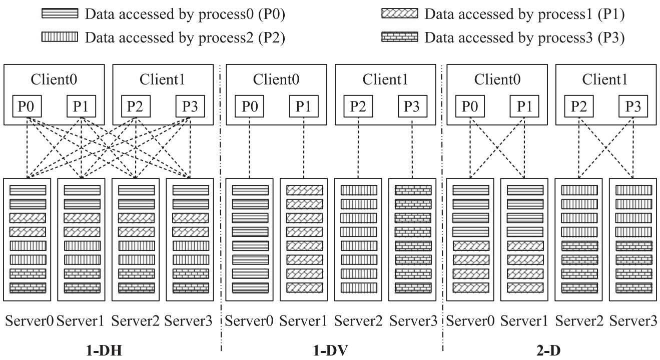


Fig. 4. Three typical data layouts in a parallel file system.

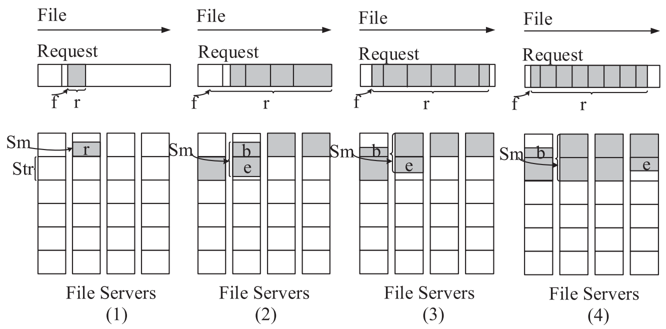


Fig. 5. Four cases where a file request involves a different number of sub-requests.

3.2.1 Access Cost of Request on HServers Under 1-DH Layout

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

$$T_H = T_s + T_t, \tag{1}$$

where T_s is the startup time, including disk seek and rotation delay, and T_t is the data transfer time spent on actual data movement.

Startup time. As a parallel request req may involve multiple sub-requests on m file servers, the startup time of req is determined by the maximum of all its sub-requests. We fist calculate the startup time of a single sub-request, then describe the startup time of the file request.

Let α denote the startup time of a sub-request on a single HServer, then α usually is a random variable since multiple sub-requests from different processes will be concurrently served by the HServer. 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 is the minimal startup time cost on an HServer, and b is the maximal startup cost.

Let X denote the startup time of request req, then it can be a variable $X = \max(\alpha_1, \alpha_2, \ldots, \alpha_m)$, where α_i ($1 \leqslant i \leqslant m$) has an independent identical distribution as α. Thus the probability density function of X is

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

TABLE 1
Parameters in Cost Analysis Model

Symbol	Description
M	Number of HServers
N	Number of SServers ($N < M$)
str	Stripe size of the parallel file system
g	Number of storage groups in 2-D layout
p	Number of processes
f	File offset of request req
r	Data size of request req
α_H	Average startup delay for HServer
β_H	Cost of access one unit of data for HServer
α_S	Average startup delay for SServer
β_S	Cost of access one unit of data for SServer

With Equation (3), the startup time of req can be calculated as

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

Under *1-DH* layout, because in the best case there is only one seek operations on each HServer, $a = \alpha_H$. But in the worst case, there are p seeks since an HServer has to concurrently serves p processes, thus $b = p * \alpha_H$. Based on the value of a and b, we can gets the cost of T_s

Data transfer time. The data transfer time T_t is determined by the maximal data transfer time of all the m sub-requests. Since each sub-request's data transfer time is proportional to its size, we first calculate the size of each sub-request, then discuss T_t for the whole file request according to the maximal sub-request size.

Under *1-DH* layout, for a given request req with offset f and size r, the serial number of the involved beginning file stripe is $B = \lfloor \frac{f}{str} \rfloor$, the ending file stripe is $E = \lfloor \frac{f+r}{str} \rfloor$, and the number of the involved file servers is

$$m = \begin{cases} E - B + 1, & E - B + 1 < M \\ M, & otherwise. \end{cases} \tag{5}$$

Accordingly, the size of the beginning fragment can be calculated as $b = str - f\%str$, and the size of the ending fragment is $e = (f + r)\%str$. Fig. 5 shows an example of the possible sub-request distributions. Let $\triangle = E - B$, $s(i)$ is the sub-request size on server i ($1 \leq i \leq m$), then $s_m = max$ $\{s(1), s(2), \ldots, s(m)\}$ can be calculated as Table 2. Based on the value of s_m, the data transfer time

$$T_t = s_m * \beta_H. \tag{6}$$

With Equations (4) and (6), T_H of each file request in Equation (1) can be obtained.

TABLE 2
The Maximal Sub-Request Sizes in Different Access Cases

Case	Maximal sub-request size (s_m)	Condition
1	r	$\triangle = 0$
2	$max\{b + e + (\lceil \frac{\triangle}{M} \rceil - 1) * str,$ $\lceil \frac{\triangle}{M} \rceil * str\}$	$\triangle > 0 \& \triangle\%M = 0$
3	$max\{b + (\lceil \frac{\triangle}{M} \rceil - 1) * str,$ $e + (\lceil \frac{\triangle}{M} \rceil - 1) * str\}$	$\triangle > 0 \& \triangle\%M = 1$
4	$\lceil \frac{\triangle}{M} \rceil * str$	otherwise

3.2.2 Access Cost of Request on SServers under Three Layouts

For each request served by SServers, we calculate the access cost in a similar way but with two differences. First, we evaluate the cost under three typical data layouts. Second, the storage parameters of SServers and HServers show distinct features in the model. On the one hand, SServer has a much smaller start up time than HServer. On the other hand, SServer has a smaller data transfer time than HServer.

For a given request req, the number of involved SServers and the maximal sub-request size are different under *1-DH*, *1-DV*, and *2-D* data layouts. We calculate the data access cost of req respectively as following.

- For *1-DH*, we assume n_h is the number of involved SServers, then the startup time

$$T_s^{1h} = a + \frac{n_h}{n_h + 1}(b - a), \tag{7}$$

where $a = \alpha_S$ and $b = p * \alpha_S$. Here we set a and b with these values because each SServer only needs one seek operations to serve a continues request in the best case and needs p startup operations to concurrently serve all the p processes in the worst case.

Assume s_{nh} is the maximal sub-request size, then the data transfer cost is

$$T_t^{1h} = s_{nh} * \beta_S. \tag{8}$$

- For *1-DV*, assume n_v is the number of involved SServers under this layout, then the startup time is

$$T_s^{1v} = a + \frac{n_v}{n_v + 1}(b - a), \tag{9}$$

where $a = \alpha_S$ and $b = \lfloor \frac{p}{N} \rfloor * \alpha_S$. Since there are $\lfloor \frac{p}{N} \rfloor$ processes on each SServer at most, we calculate b differently compared to *1-DH* layout.

Similarly, assume s_{nv} is the maximal sub-request size, then the data transfer cost of req is

$$T_t^{1v} = s_{nv} * \beta_S. \tag{10}$$

- For *2-D*, we assume n_g is the number of involved SServers, then the startup time of req can be calculated as

$$T_s^{2d} = a + \frac{n_g}{n_g + 1}(b - a), \tag{11}$$

where $a = \alpha_S$ and $b = \lfloor \frac{p}{g} \rfloor * \alpha_S$. We set b with this value because there are $\lfloor \frac{p}{g} \rfloor$ processes on one SServer at most in this case.

For the data transfer cost, assume s_g is the maximal sub-request size, then it can be described as

$$T_t^{2d} = s_{ng} * \beta_S. \tag{12}$$

Based on Equation (7) to Equation (12), we can get the data access costs of req on SServers under three data layouts, denoted by T_S^{1h}, T_S^{1v}, and T_S^{2d}, which are the sum of the corresponding startup time and transfer time respectively.

CDT

...				
D_file	D_offset	Length	Layout	C_flag
D_file	D_offset	Length	Layout	C_flag
...				

DMT

...					
D_file	D_offset	C_file	C_offset	Length	D_flag
D_file	D_offset	C_file	C_offset	Length	D_flag
...					

Fig. 6. The data structure of CDT and DMT.

The above formulas show that more processes will lead to a larger access cost whatever the data layout is. This is because a larger p produces a longer startup time, implying that the model could dynamically change under I/O interference from multiple processes.

3.3 Layout-Aware Critical Data Identification

With the proposed data access cost model, the available performance benefit of request req on SServers under three data layouts can be calculated as following:

$$B^{1h} = T_H - T_S^{1h}, \tag{13}$$

$$B^{1v} = T_H - T_S^{1v}, \tag{14}$$

$$B^{2d} = T_H - T_S^{2d}. \tag{15}$$

For the given request req, each layout may incur a different access cost. Although an HServer has lower performance than an SServer, all HServers in a parallel environment can provide higher I/O performance if more HServers are deployed. Thus, the performance benefit of each layout is not always positive.

Among the three layouts, the optimal data layout yields the lowest access cost and brings the maximal performance benefit, hence the data should be placed on SServers with this optimal layout if needed. Assume the maximal performance benefit of req is B, then B can be described as

$$B = \max\{B^{1h}, B^{1v}, B^{2d}\}. \tag{16}$$

To maximize system performance, *Data Identifier* identifies the performance-critical data based on the value of B. A positive B means that serving the request on SServers will reduce the I/O access time, i.e., increase the I/O system performance, thus the request should be critical and served on SServers with the optimal layout. Otherwise, serving the request on HServers helps improve the I/O performance and there is no need to serve it on SServers. Therefore, if B of a request is larger than zero, *Data Identifier* regards the requested data as performance-critical data, and selectively caches it on SServers with the optimal data layout bringing maximal performance benefit.

Previous work identifies performance-critical data only based on the value of B^{1h} [8]. If B^{1h} is larger than zero, the data is regarded as performance-critical and should be cached on SServers. As only default *1-DH* layout is considered, this layout-oblivious identification scheme may suffer from sub-optimal performance as shown in Section 5.

Data Identifier uses a critical data table (CDT) to record the information of performance-critical data. As shown in Fig. 6, each entry in CDT consists of five variables, D_file, D_offset, Length, Layout, and C_flag, indicating the original file name on HServers, the data offset in the file, the data length, the optimal data layout, and whether the data needs to be cached on SServers, respectively. It is worth noting that, while the performance-critical data are eligible, they may not be in the cache system. The information of the data really in the cache is managed by another data table DMT, which we will discuss in the next section.

3.4 Cache Metadata Management

3.4.1 Kernel Metadata Structure

SLA-Cache uses a data mapping table (DMT) to keep track of data information that has been cached on SServers. To enable layout-aware data placement on SServers, SLA-Cache creates three cache files, each using a different layout, for each original user file. As shown in Fig. 6, each entry in DMT includes six important fields. D_file and D_offset are the file name and offset for the data in the original file, C_file and C_offset are the file name and offset for the data in the cache file. Length is the size of the cached data, and D_flag indicates whether the cached data is dirty. The D_flag is set when SServers contains data that requires to be copied back to HServers. DMT is updated each time a data location has changed. By maintaining DMT, *Redirector* can continuously track the most up-to-date location of the data, which ensures data consistency between HServers and SServers.

In memory, DMT is organized as a hash table to speedup lookups, incurring minimal overhead with several memory accesses. Since only remapped data needs to be tracked, the spatial overhead of the mapping table is small. Besides the memory-resident copy, the DMT table is also maintained in persistent storage. In order to reduce the I/O delay of DMT access, in our implementation, DMT is written to an addressable file on SServers. Changes to the mapping table are synchronously written to the storage in order to survive power failures.

3.4.2 Metadata Consistency

In a parallel I/O environment, there are multiple processes possibly accessing DMT concurrently. To keep the consistency of the metadata, DMT is maintained in a global data file, and each process sends a lock request to access the DMT table. To simplify the implementation, we leverage *Berkeley DB* to perform metadata operations and address lock contention. Similar techniques, such as the distributed metadata management [11], can also be applied to maintain the metadata among multiple processes, to minimize the communication contention of metadata accesses.

3.5 Selective and Layout-Aware Caching Scheme

Redirector is a core module in SLA-Cache, it caches data on SServers based on four factors: (1) the mapping entry in DMT, indicating if the request can be served by SServers, (2) the entry in CDT, indicating if the missed request should be admitted on SServers and what is the preferred layout to store the request data, (3) type of I/O request (read or write), and (4) the available space on SServers.

When *Redirector* receives an I/O request, it looks up DMT and checks if the request hits SServers or not. If so, *Redirector* directly serves the request with the data in SServers. Otherwise, *Redirector* handles the request obeying a selective cache admission policy. If the request is a write operation, SLA-Cache stores the data on SServers with a layout-aware cache placement policy.

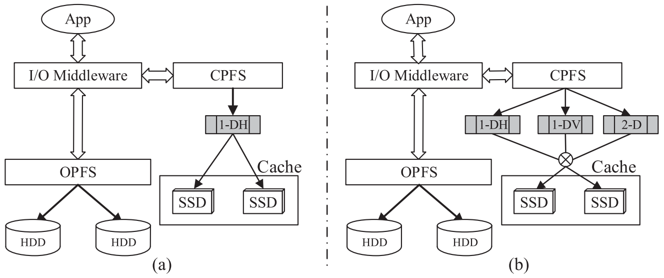


Fig. 7. (a): Current cache systems are layout-oblivious to place cache data. (b) The cache data placement in SLA-Cache.

Algorithm 1. Redirection Algorithm

Require: I/O Request: *req*, Data Mapping Table: DMT, Critical Data Table: CDT.

1: **if** *req* misses in DMT **then**
2:   **if** *req* is write **then**
3:     **if** *req* is in CDT **then**
4:       find free space on SServers
5:       **if** free space is found **then**
6:         allocate space for *req*
7:         set the space layout with the *layout* in CDT
8:         add new entry in DMT (mark dirty)
9:         change the *req* location as the DMT entry
10:       **else**
11:         find clean space on SServers
12:         **if** clean space is found **then**
13:           allocate space for *req*
14:           set the space layout with the *layout* in CDT
15:           change the entry in DMT (mark dirty)
16:           change the *req* location as the DMT entry
17:         **end if**
18:       **end if**
19:     **end if**
20:   **else**
21:     **if** *req* is in CDT **then**
22:       set the C_flag of the entry in CDT
23:     **end if**
24:   **end if**
25: **else**
26:   change the *req* location as the DMT entry
27: **end if**
28: send request *req*

Algorithm 1 shows the work-flow of *Redirector* for each I/O request. The algorithm attempts to utilize SServers whenever possible. For write requests, SServers are regarded as a write buffer. If there is a sufficient space in SServers (lines 5 and 12) or the request is already mapped (line 26), the request will be absorbed by SServers. To reduce data migration, the algorithm first looks for free space in SServers when allocating an available space for a write request. If the free space cannot be found, a clean space will be the candidate based on a LRU policy. To further improve performance, the new write data are stored on SServers with the optimal data layout specified by the *layout* field in CDT (lines 7 and 14), which could be *1-DH*, *1-DV*, or *2-D*. Fig. 7 illustrates the cache data placement of SLA-Cache, compared with the layout-oblivious cache systems [8]. Current parallel file systems provide interfaces to set the layout attributes of a file directory. To enable the layout-aware cache placement, SLA-Cache creates three cache files, each in a different directory configured with a specific data layout, for an original data file. By writing the data into a different target cache file, the new write data can be stored on SServers with an optimal data layout.

For read requests, *Redirector* uses SServers as a caching area. When the required data misses, the request is cached in a "lazy" way. This means that Redirector marks the C_flag in the corresponding entry of CDT (line 22), which indicates that an actual data movement should be conducted by *Rebuilder* in the following data reorganization stage. This method reduces the response time of read requests.

Please note that this algorithm is selective: instead of writing or reading all data, it only attempts to absorb the most performance-critical requests in CDT (lines 3 and 21), to efficiently utilize SServers space. Furthermore, this algorithm is layout-aware: instead of writing data on SServers only with the default *1-DH* layout, it stores data with an optimal data layout requiring the lowest access cost among three candidates, to further improve the cache system performance.

3.6 Data Reorganization

Rebuilder plays the role of freeing SServers space for future use. It is triggered periodically, and performs two kinds of operations. 1) It writes dirty data back to HServers, and then sets the D_flag in DMT to 0, indicating the data is clean and the space is available for future use. 2) It reads data from the HServers into SServers by consulting the CDT table, and then sets the C_flag to 0 to show the data has been cached.

The data reorganization activities may interfere with the normal I/O activities. For this reason, *Rebuilder* issues low-priority I/O requests for the reorganization to reduce the interference to the normal I/O operations.

4 Implementation

We have implemented a prototype of SLA-Cache under MPICH2 [23]. The primary and challenging parts are explained below.

4.1 Cache Metadata Mapping Table

Both *Redirector* and *Rebuilder* need to get application data access information from DMT. DMT is a key structure to store the mapping relationship between the data cached on SServers and HServers.

We use *Berkeley DB* [24] to implement the DMT table. DMT is a database file which has a standalone space on SServers. The *Berkeley DB* is configured as a hash table, and each record is a key-value pair. We generate a mapID by encoding the original file name (including its full path). Each record in the *Berkeley DB* hash table is a key-value pair; the key is the mapID and the value contains the data access information listed in Fig. 6. By leveraging the light-weighted database, the lock contention is addressed and metadata operations are performed efficiently. Additionally, we also use a list to maintain the most frequently accessed mapping entries which further reduces the in-memory mapping table size.

4.2 I/O Redirection Module in MPI-IO

The I/O redirection module redirects data accesses on the original files to the cache files. Usually an application issues a data request with three parameters: the identifier of the original file, the data offset, and the request size. The redirection module translates the filename and offset between the original file and the cache file and serves the request using the cache file. We have made the following modifications to the standard MPI-IO functions.

MPI_File_open: While opening a file, in addition to open the original file, the method also opens three corresponding cache files, each with a different layout.

MPI_File_read: For each I/O read, this method first checks whether the opened cache files contains the requested content by looking up DMT. If it is true, the module calculates the correct data offset, and issues the data request using the new offset and the cache file handle. Otherwise, the module gets the data using the original file handle and offset. At the same time, this module uses the input parameters to calculate the performance benefits with Equation (13) to Equation (15). If the request is critical and not in CDT, the method adds it to CDT with a new entry, and sets the C_flag that will be used by *Rebuilder* later.

MPI_File_write: For each I/O write, this module checks whether the opened cache files contains the requested content by initiating a lookup in DMT. If this is true, one cache file will be found, and the module calculates the correct data offset, and issues the data request using the new offset and cache file handle. Otherwise, the module determines whether the access data belongs to CDT. If so, the method tries to allocate available space from the cache file with the optimal data layout for the critical write request, updates the DMT entry, and issues a data request with the new offset and cache file handle. Otherwise, the module writes the data using the original file handle and offset.

MPI_File_close: In addition to the original file, it also closes the opened cache files.

MPI_File_seek: It calculates the offset and conducts the seek operation in the cache files.

When the requested data does not belong to any cache file and is not performance-critical, this system will act the same as the default MPI-IO implementation.

4.3 Data Movement Implementation Issues

In order to avoid interfering with the normal MPI I/O operations, *Rebuilder* 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. Each process can have multiple files opened, but only one thread is created. Once the I/O thread is created, it enters an infinite loop to perform the data movement operation 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.

5 Performance Evaluation

5.1 Experimental Setup

The experiments are conducted on a 65-node SUN Fire Linux cluster. Each computing node has two AMD Opteron

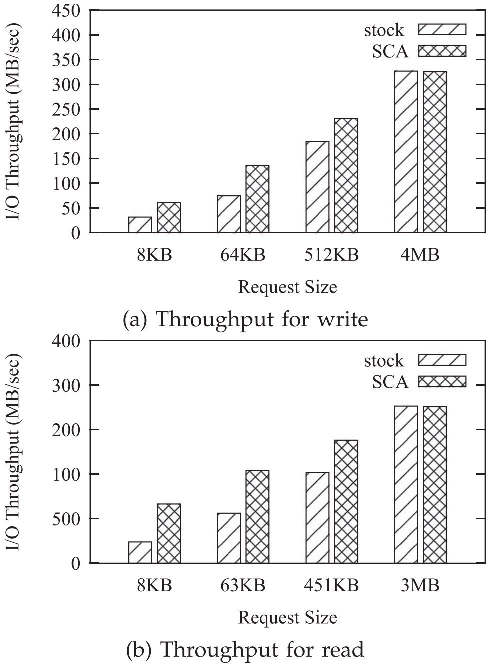


Fig. 8. I/O throughputs of IOR with varied request sizes.

processors, 8 GB memory and a 250 GB HDD (SEAGATE ST32502NSSUN250G). The operating system is Ubuntu 9.04 and the parallel file system is PVFS2 version 2.8.2. All nodes are equipped with Gigabit Ethernet interconnection, and eight nodes are equipped with an additional PCI-E X4 100 GB SSD (OCZ-REVODRIVE X2). Although a more high-end SSD would certainly improve cache performance, this entry-level SSD well demonstrates the effectiveness and potential of SLA-Cache.

Among the available nodes, 32 nodes are used as computing nodes, eight are HServers, and four are SServers. Each HServer uses one HDD as the storage device and each SServer uses one SSD. HServers and SServers are separately accessed with two PVFS2 parallel file system. MPICH2 [23] compiled with ROMIO is used to generate the executable. When SLA-Cache is enabled, the cache capacity is set to 30 percent of the application's data size. SLA-Cache does not benefit read performance if the requested data have not been cached on SServers. However, many MPI programs are executed several times and present consistent data access patterns [18], [25]. The critical data identified and cached by SLA-Cache in the first run can improve read performance in the later runs. Therefore, the read performance improvement of SLA-Cache for the program with a second run is shown in this paper.

We use the popular parallel file system benchmark IOR [20], HPIO [26], and MPI-Tile-IO [27] to conduct the experiments. In our experiments, each data point is an average of five trials.

5.2 Evaluation on Selective Cache Admission

We first show the effectiveness of the selective cache admission policy of SLA-Cache in improving the original I/O system (i.e., the stock I/O system) performance. To ensure that the improvements only come from applying SCA, we make data caching decisions based on the value of B^{1h} instead of B as in [8], and disable the layout-aware cache placement.

TABLE 3
Performance Comparison of SCA and NSCA

Req Size	SCA (MB/s)	NSCA(MB/s)
8 KB	60.6	56.1
64 KB	136.1	127.3
512 KB	230.9	218.4
4 MB	325.2	293.5

Therefore, the system only uses *1-DH* layout to distribute cache data on SServers with a stripe size of 64 KB.

5.2.1 The IOR Benchmark

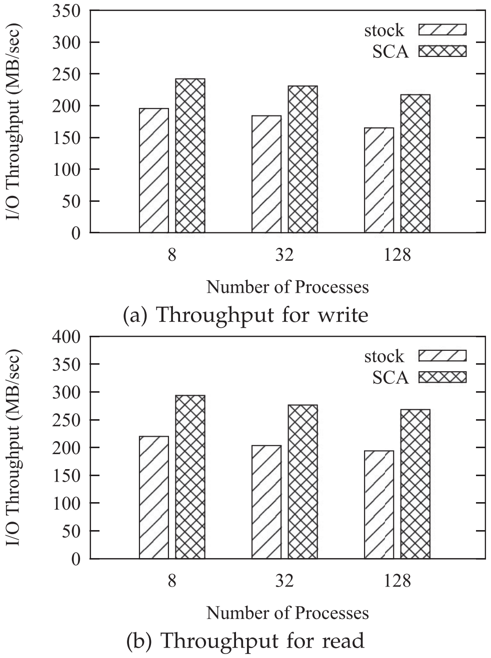
IOR is a parallel file system benchmark developed at Lawrence Livermore National Laboratory [20]. It provides three APIs: MPI-IO, POSIX, and HDF5, we only use MPI-IO in the tests. During these benchmarks, 32 processes are used and the request size is kept to 512 KB unless otherwise specified.

We first run IOR with varying request size from 8 KB to 4 MB. Each process issues random I/O requests and accesses 512 MB of data. As shown in Fig. 8a, SCA can improve the stock I/O system throughput by 92.1, 82.5, and 25.4 percent for writes with the request size of 8, 64, and 512 KB respectively. With smaller request sizes, the I/O throughput improvement is more significant because SServers can lead to more benefits for small random requests. We also note that, for request size of 4 MB, SCA nearly has the same I/O throughput as the stock I/O system. As HServers have higher I/O parallelism and the performance gap between SServer and HServer is reduced for large requests, placing them on SServers incurs less or no performance benefits. Thus, SCA can bring less performance improvement. The read test yields similar results, as shown in Fig. 8b. We can see that, the overall throughput is increased by up to 178.5 percent with the request size of 8 KB. Compared to write requests, SCA has larger read performance improvements because SSD performs better for reads than writes.

To verify the effectiveness of the selective policy, we also test the system write performance with a *non-selective* counterpart (NSCA), where all requests are admitted to SServers indiscriminately. Table 3 shows the performance comparison of SCA and NSCA. We can see NSCA has lower performance than SCA. Especially, for request size of 4 MB, NSCA is even worse than the original stock system, meaning that admitting all requests into SServers non-selectively can significantly degrade the system performance. As opposed to NSCA, SCA can effectively identify the performance-critical data and redirect them to SServers for better performance.

Next, to examine the impact of the number of processes, we run IOR with 8, 32, and 128 processes. Fig. 9a gives the results of write requests. Similar to the previous test, SCA improves the overall I/O throughput by 23.9 to 31.7 percent. With the increase of process number, IOR's throughput gets lower because each HServer and SServer needs to serve more requests and the competition among processes gets more severe. This result also shows SCA has a good scalability in terms of the number of processes. The performance trend is similar for reads, as shown in Fig. 9b.

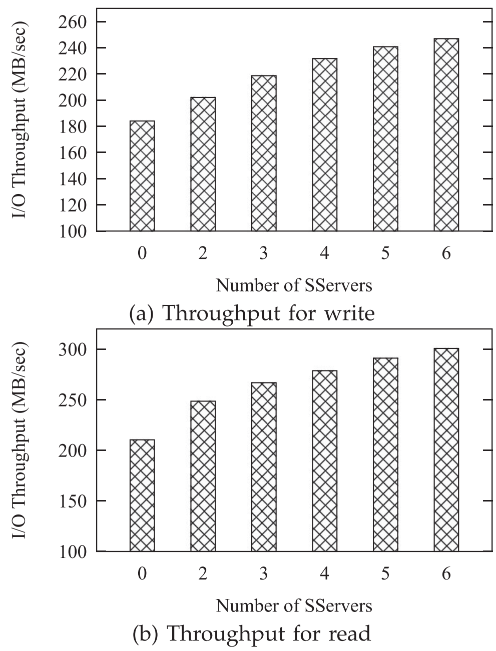
Then we vary the capacity of each SServer to examine the sensitivity of the system to the available caching space. In general, the overall capacity of SServers is much smaller



(a) Throughput for write

(b) Throughput for read

Fig. 9. I/O throughputs of IOR with varied numbers of processes.



(a) Throughput for write

(b) Throughput for read

Fig. 10. I/O throughputs for the IOR benchmark with varied numbers of SServers.

than that of HServers and could be even smaller than the I/O working set size for the application. According to the algorithm, SCA could elastically replace the cached data to increase the utilization of the SServer space. Table 4 shows the write throughputs when the capacity is varied from 0 to 6 GB. Here 0 GB means that SCA is disabled. It is observed that I/O throughput improves by increasing the capacity of SServers, which is because more performance-critical I/O requests can benefit from SServers. However, when most these requests are already cached, continuously enlarging SServers will only bring limited performance improvement.

Finally we run IOR with different number of SServers. We set the number from zero to six. Zero means the stock I/O system is used. We keep the cache size to 30 percent of the application's data size. Fig. 10a shows the results for write operations. The overall write throughput is improved by 9.7 to 34.2 percent. As the number of SServers increases, the I/O throughput improves because more SServers provides higher I/O parallelism and thus can serve the redirected requests with better I/O performance. However, the improvement reduces when continuously adding more nodes to SServers. This is because only a portion of the I/O workload is redirected to SServers and the improvement is bounded to these requests. Hence, choosing a reasonable number of file servers based on the characteristic of the I/O workload is critical to make full use of SServers. For reads, SCA has a higher I/O throughput than writes due to the better read performance of SSD in each SServer, as shown in Fig. 10b.

TABLE 4
I/O Throughputs of IOR with Varied Space of SServers

Space	Throughput (MB/s)	Speedup (%)
0 GB	184.1	0
2 GB	211.3	14.7
4 GB	227.4	23.5
6 GB	242.5	31.6

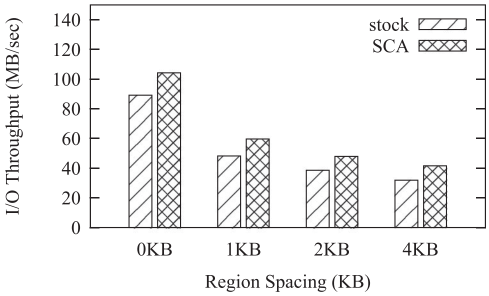
5.2.2 The HPIO Benchmark

HPIO is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate I/O [26]. This benchmark can generate various data access patterns by changing three parameters: region count, region spacing, and region size. The region spacing is used to generate noncontiguous data access patterns. In our experiment, the number of process is 32, the region count is 4,096, the region size is 16 KB, and the region spacing is varied from 0 to 4 KB (0 KB indicates sequential access).

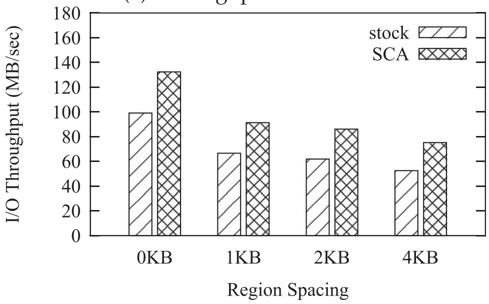
Fig. 11a shows the results of write requests. SCA can increase the I/O throughput by 16.9, 23.7, 24.2, and 30.3 percent respectively. It means that SCA is effective with respect to HPIO benchmark. We also note that, as the region spacing increases, the performance speedup gets more obvious. This is because noncontiguous I/O requests can benefit more from SServers than HServers. However, though the I/O access of each process is noncontiguous, it is not as random as the IOR benchmark, thus the improvements for HPIO are not as significant as those for IOR. This also confirms the adaptability of SCA; when the application's I/O accesses have a poorer throughput (due to the poorer data sequential locality among consecutive accesses), more benefit is gained by using SCA. For read operations, the performance has similar trend as presented in Fig. 11b.

5.2.3 The MPI-Tile-IO Benchmark

MPI-Tile-IO is a test application from the Parallel I/O Benchmarking Consortium [27]. It treats the entire data file as a two-dimensional dense dataset and tests the performance of noncontiguous data access patterns. Each process accesses a chunk of data based on the size of each tile and the size of each element. In the tests, the number of elements



(a) Throughput for write

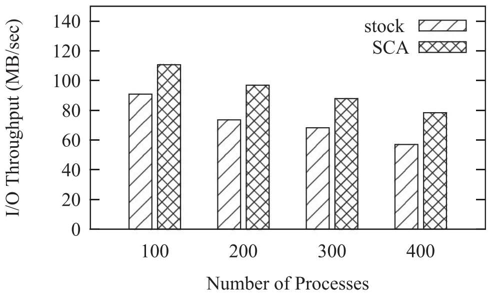


(b) Throughput for read

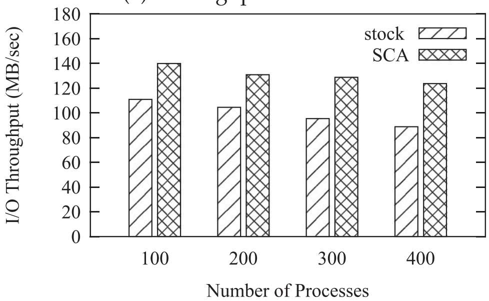
Fig. 11. I/O throughputs of HPIO with varied region spacings.

in the X and Y directions are set to 20 and 20, the size of each element is set to 64 KB, and the number of processes is varied between 100 and 400.

Fig. 12 shows the aggregated I/O throughputs. The aggregated throughput increases by 21.7 to 37.6 percent for writes, and 26.2 to 39.3 percent for reads. As mentioned above, the data access patterns of MPI-Tile-IO are nested-stride. This means, each process has a fixed-stride access pattern and yields better data locality than that of the IOR tests. As a result, the performance improvement of this benchmark is not as large as that of IOR, but is still significant. This further confirms that SCA brings additional benefits when data requests are more random in nature.



(a) Throughput for write



(b) Throughput for read

Fig. 12. I/O throughputs of MPI-Tile-IO with varied numbers of processes.

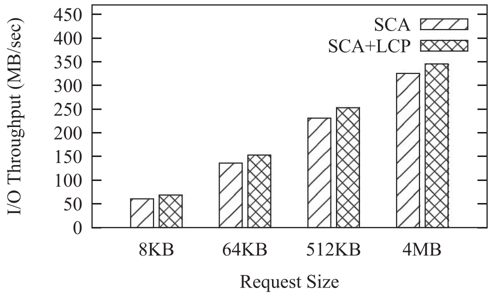


Fig. 13. I/O throughputs of IOR with varied request sizes.

5.3 Evaluation on Layout-Aware Cache Placement

We conduct experiments to show that the layout-aware cache placement policy can further improve I/O system performance, which verifies the need to optimize the data layout of the cache data. We store the cache data in two different ways, one set of them are stored in PVFS's default data layout (*1-DH*) as in SCA, and the other set of data are stored in the optimal data layout determined by the cost model presented in Section 3.2. These two sets of data are logically identical with each other, so all the performance differences are owing to the differences of the physical data layouts.

5.3.1 The IOR Benchmark

We vary the request size of IOR. We run IOR with request sizes of 8, 64, 512 KB, and 4 MB. The process number is fixed to 32. The corresponding results for writes are shown in Fig. 13. We can see that LCP obtains 6.3-13.2 percent additional performance improvements based on the performance that is already greatly boosted by SCA with the request size of 8, 64 and 512 KB. When the request size is 4 MB, LCP achieves 5.8 percent improvement over the stock I/O system while only SCA can not improve the stock I/O system performance. This is because the optimal data layout makes better I/O performance of SServers.

We also vary the number of processes. We run IOR with 8, 32, and 128 processes, and set the request size to 512 KB. Fig. 14 describes the results of write requests. LCP obtains 11.5-18.8 percent extra performance improvements over the system where only SCA is applied. SLA-Cache can improve the stock I/O system performance by 38.2 to 56.6 percent by applying both SCA and LCP.

As read tests show similar trends, we do not give the results of read tests.

5.3.2 The HPIO Benchmark

We set the number of processes to 32, the region count to 4,096, and the region size to 16 KB. We vary the region

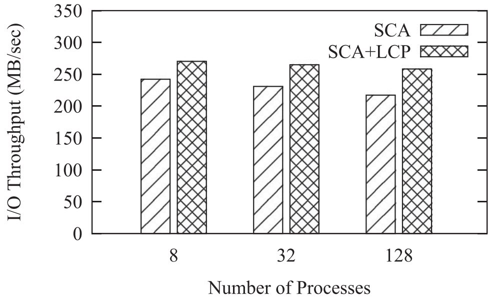


Fig. 14. I/O throughputs of IOR with varied numbers of processes.

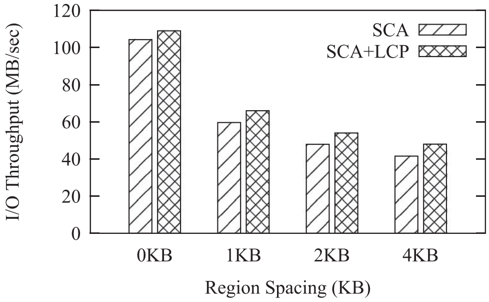


Fig. 15. I/O throughputs of HPIO with varied region spacings.

spacing from 0 to 4 KB. Fig. 15 shows the results. Because the most used optimal layout is *1-DH*, as adopted by SCA, the additional improvements obtained by LCP are not very large. However, LCP still can further improve SCA by 4.5, 5.6, 6.7, and 9.5 percent respectively. Compared to the stock system, SLA-Cache that applying both SCA and LCP can increase the overall I/O throughput by 22.3, 30.6, 32.2, and 42.6 percent respectively. It means that SLA-Cache is effective with respect to HPIO benchmark.

5.3.3 The MPI-Tile-IO Benchmark

In the tests, we configure the number of elements in the X and Y directions as 20 and 20, the size of each element as 64 KB. We vary the number of processes between 100 and 400. Fig. 16 shows the aggregated I/O throughputs. With LCP, the I/O performance can be further improved by 10.2, 11.4, 12.6, and 21.2 percent respectively. For larger number of processes, the improvement is more significant because *1-DV*, which is the commonly used optimal layout in LCP, can bring more performance benefits for requests with high concurrency (number of processes). Compared to the stock system, SLA-Cache can increase the overall I/O throughput by 34.2, 45.4, 46.8, and 66.7 percent respectively. This means that the layout-aware cache placement is effective with respect to MPI-Tile-IO benchmark.

5.4 System Overhead

5.4.1 Metadata Space Overhead

To track data cached on SServers and maintain data consistency, SLA-Cache uses a file in SServers to store the DMT Table, incurring additional storage space overhead.

The system has a maximal space overhead when all the requests are of 4 KB. Assuming the available storage space of SServers is S GB, each entry in our implementation occupies $6 * 4$ B, then the maximal number of records in DMT is $S/4 * 10^6$. Therefore, the maximal metadata space overhead is 0.6 percent of the SServer space, which is negligible.

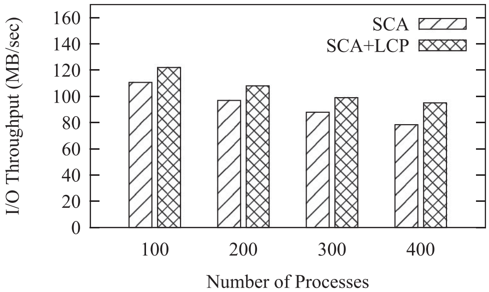


Fig. 16. I/O throughputs of MPI-Tile-IO with varied numbers of processes.

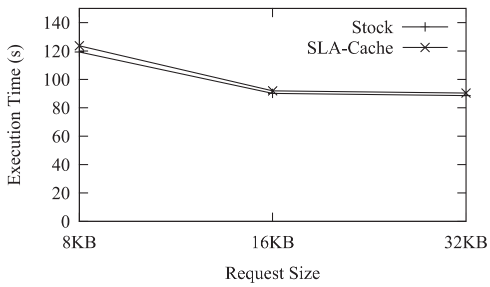


Fig. 17. Performance overhead result.

5.4.2 Performance Overhead

As shown in Fig. 3, SLA-Cache has some additional modules which may generate performance overhead. SLA-Cache is able to improve the I/O performance of applications with performance-critical requests. However, it may degrade the I/O performance for some applications do not have performance-critical requests. Thus it is necessary to evaluate the following two possible sources of overhead during runtime.

1) During file open operation, *Data Identifier* module needs to initialize the DMT table in memory, and decide whether to create a new cache file in SServers.

2) During file read/write operation, *Data Identifier* and *Redirector* need to calculate the access cost, perform a lookup in CDT and DMT, and decide whether to cache the requested data. Since DMT has been loaded from SServers, most of the operations can be done in memory.

The performance overhead mentioned above is very small compared with I/O access overhead. To show this, we run IOR with request size from 8 to 32 KB. The process number is 32, and each process writes a shared 10 GB file in a random pattern where all the requests intentionally miss SServers. This causes *Redirector* to redirect all requests to HServers. Fig. 17 shows the results. As expected, the performance overhead is negligible.

6 Related Work

6.1 I/O Request Stream Optimization

A lot of efforts have focused on I/O request reorganizations to address small data access issue in I/O middleware. For multiple noncontiguous smaller requests, Data sieving [9] technique integrates them into a larger contiguous chunk including the additional data (hole) instead of accessing them separately. Datatype I/O [28] and List I/O [10] techniques allow users to merge multiple I/O requests with different patterns within a single I/O routine. Collective I/O [9], [29] is another technique proposed to rearrange concurrent I/O accesses among a group of processes of a parallel program to a larger contiguous request.

All these techniques succeed in exploiting regular group relation for parallelism, but they are not designed to utilize SSDs for random access. SLA-Cache can use not only these techniques for its underlying parallel file systems but also utilize SSDs' characteristics.

6.2 Using System Memory as a Cache

Traditionally the problem of I/O accesses is addressed by using system main memory as a cache. These schemes are deployed on both client side and server side in a parallel environment, including client-side file caching in GPFS [5] and Lustre [4], cooperative caching [30], active buffering [12] and collective caching [11].

In contrast to these memory-based methods, SLA-Cache has larger cache capacity and is reliable due to its use of non-volatile SSDs. SSDs are a complement of memory cache and can be served as an extension of memory cache. However, SLA-Cache has a totally different selection algorithm and runtime system design. The integration of memory cache and SLA-Cache will be an interesting topic for future study.

6.3 SSD-Based Storage System

Using SSDs as a cache of traditional HDDs is a widely used strategy in I/O systems, such as FlashCache [15], Conquest [31], Burst Buffer [32], and LADS [33]. Tiered checkpointing redirects all write data to the RAM disks or SSDs in the computing nodes [34]. SSD-based hybrid storage is another popular method to make full use of SSDs. This method integrates an SSD and a hard disk as one block device [35], [36]. I-CASH is a new hybrid storage architecture based on data-delta pairs to improve I/O performance for I/O-intensive workloads [17]. Hystor identifies critical data blocks with strong temporal locality and redirects them to SSD for fast future accesses [16].

These approaches succeed in exploiting data access information *within a single file server or a single computing node*. But unlike this work, SLA-Cache leverages the global data access information *in a parallel I/O system* to improve performance. Our previous work [37], [38], [39], [40], [41], [42] similarly uses the global data information and SSDs in a parallel I/O environment. However, the SSD-based servers are used as persistent storage instead of a cache. With a small set of SSD-based file servers and the selective cache admission and layout-aware cache placement policy, SLA-Cache provides a feasible and cost-effective solution for large-scale data intensive applications.

7 Conclusions

In this study, we introduced a Selective and Layout-Aware SSD Cache (SLA-Cache) system to improve the performance of a parallel I/O system. SLA-Cache deploys a small set of SSD-based file servers as a cache of conventional HDD-based file servers. To make full utilization of the limited space of SSD-based file servers, SLA only admits performance-critical data which are identified by a cost model. Furthermore, SLA-Cache stores data on SSD-based file servers in a layout-aware style to further improve I/O performance. We have implemented a prototype of SLA-Cache under MPICH2. The performance of SLA-Cache is evaluated with different benchmarks, namely IOR, HPIO, and MPI-TILE-IO, on a SSD-equipped computer cluster. Experimental results show that SLA-Cache is feasible and effective in improving parallel I/O performance.

Acknowledgments

This work is supported in part by the National Basic Research 973 Program of China under Grant No. 2015CB352400, the National Science Foundation of China under Grant No. 61572377, the Natural Science Foundation of Hubei Province of China under Grant No. 2014CFB239, the Open Fund from HPCL under Grant No. 201512-02, the Open Fund from SKLSE under Grant No. 2015-A-06, and the US National Science Foundation under Grant CNS-1162540.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA, US: Morgan Kaufmann, 2011.
[2] M. Kandemir, S. W. Son, and M. Karakoy, “Improving I/O performance of applications through Compiler-directedcode restructuring,” in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 159–174.
[3] P. H. Carns, I. Walter B. Ligon, R. B. Ross, and R. Thakur, “PVFS: A parallel virtual file system for linux clusters,” in *Proc. 4th Annu. Linux Showcase Conf.*, 2000, pp. 317–327.
[4] S. Microsystems, “Lustre file system: High-performance storage architecture and scalable cluster file system,” Lustre File System White Paper, 2007.
[5] F. Schmuck and R. Haskin, “GPFS: A shared-disk file system for large computing clusters,” in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, pp. 231–244.
[6] X. Zhang, K. Davis, and S. Jiang, “iTransformer: Using SSD to improve disk scheduling for high-performance I/O,” in *Proc. 26th IEEE Int. Parallel Distrib. Process. Symp.*, 2012, pp. 715–726.
[7] X. Zhang, K. Liu, K. Davis, and S. Jiang, “iBridge: Improving unaligned parallel file access with solid-state drives,” in *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 381–392.
[8] S. He, X.-H. Sun, and B. Feng, “S4D-Cache: Smart selective SSD cache for parallel I/O Systems,” in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 514–523.
[9] R. Thakur, W. Gropp, and E. Lusk, “Data sieving and collective I/O in ROMIO,” in *Proc. 7th Symp. Frontiers Massively Parallel Comput.*, 1999, pp. 182–189.
[10] A. Ching, A. Choudhary, K. Coloma, L. Wei-keng, R. Ross, and W. Gropp, “Noncontiguous I/O accesses through MPI-IO,” in *Proc. 3rd IEEE/ACM Int. Symp. Cluster Comput. Grid*, 2003, pp. 104–111.
[11] W.-k. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tidemad, “Collective caching: Application-aware Client-side file caching,” in *Proc. 14th IEEE Int. Symp. High Perform. Distrib. Comput.*, 2005, pp. 81–90.
[12] X. Ma, M. Winslett, J. Lee, and S. Yu, “Improving MPI-IO output performance with active buffering plus threads,” in *Proc. Int. Parallel Distrib. Process. Symp.*, 2003, pp. 68–77.
[13] Y. Xu and S. Jiang, “A scheduling framework that makes any disk Schedulers non-work-conserving solely based on request characteristics,” in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, p. 9.
[14] A. Caulfield, L. Grupp, and S. Swanson, “Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications,” in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Operat. Syst.*, 2009, pp. 217–228.
[15] M. Srinivasan and P. Saab. (2013). Flashcache: A general purpose writeback block cache for linux, [Online]. Available: https://github.com/facebook/flashcache
[16] F. Chen, D. A. Koufaty, and X. Zhang, “Hystor: Making the best use of solid state drives in high performance storage systems,” in *Proc. Int. Conf. Supercomput.*, 2011, pp. 22–32.
[17] Q. Yang and J. Ren, “I-CASH: Intelligently coupled array of SSD and HDD,” in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 278–289.
[18] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur, “Pattern-direct and Layout-aware replication scheme for parallel I/O systems,” in *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 345–356.
[19] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, “Parallel I/O prefetching using MPI file caching and I/O signatures,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Analysis*, 2008, pp. 1–12.

[20] (2016). Interleaved Or Random (IOR) Benchmarks [Online]. Available: http://sourceforge.net/projects/ior-sio/
[21] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 37–48.
[22] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proc. 6th Workshop I/O Parallel Distrib. Syst.*, 1999, pp. 23–32.
[23] (2016). A. N. Lab. MPICH2: A high performance and widely portable implementation of MPI [Online]. Available: http://www.mcs.anl.gov/research/project-detail.php?id=2
[24] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proc. USENIX Annu. Tech. Conf.*, 1999, pp. 183–191.
[25] Y. Wang and D. Kaeli, "Profile-guided I/O partitioning," in *Proc. 17th Annu. Int. Conf. Supercomput.*, 2003, pp. 252–260.
[26] A. Ching, A. Choudhary, W.-K. Liao, L. Ward, and N. Pundit, "Evaluating I/O characteristics and methods for storing structured scientific data," in *Proc. 20th Int. Parallel Distrib. Process. Symp.*, 2006.
[27] (2016). MPI-Tile-IO Benchmark [Online]. Available: http://www.mcs.anl.gov/research/projects/pio-benchmark/
[28] A. Ching, A. Choudhary, W.-k. Liao, R. Ross, and W. Gropp, "Efficient structured data access in parallel file systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2003, pp. 326–335.
[29] Y. Chen, S. Xian-He, R. Thakur, P. C. Roth, and W. D. Gropp, "LACIO: A new collective I/O strategy for parallel I/O systems," in *Proc. 25th IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 794–804.
[30] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proc. 1st USENIX Conf. Operat. Syst. Des. Implementation*, 1994.
[31] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better performance through a Disk/persistent-RAM hybrid file system," in *Proc. USENIX Annu. Tech. Conf.*, 2002, pp. 15–28.
[32] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.
[33] Y. Kim, S. Atchley, G. R. Valle, and G. M. Shipman, "LADS: Optimizing data transfers using layout-aware data scheduling," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 67–80.
[34] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable Multi-level checkpointing system," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2010, pp. 1–11.
[35] H. Payer, M. Sanvido, Z. Bandic, and C. Kirsch, "Combo drive: Optimizing cost and performance in a heterogeneous storage device," in *Proc. 1st Workshop Integr. Solid-State Memory Storage Hierarchy*, 2009, vol. 1, pp. 1–8.
[36] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, "HybridStore: A cost-dfficient, high-performance storage system combining SSDs and HDDs," in *Proc. IEEE 19th Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, 2011, pp. 227–236.
[37] S. He, X.-H. Sun, B. Feng, X. Huang, and K. Feng, "A cost-aware region-level data placement scheme for hybrid parallel I/O systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2013, pp. 1–8.
[38] S. He, X.-H. Sun, B. Feng, and F. Kun, "Performance-aware data placement in hybrid parallel file systems," in *Proc. 14th Int. Conf. Algorithms Archit. Parallel Process.*, 2014, pp. 563–576.
[39] S. He, Y. Liu, and X.-H. Sun, "A performance and space-aware data layout scheme for hybrid parallel file systems," in *Proc. Data Intensive Scalable Comput. Syst. Workshop*, 2014, pp. 41–48.
[40] S. He, X.-H. Sun, and A. Haider, "HAS: Heterogeneity-aware selective data layout scheme for parallel file systems on hybrid servers," in *Proc. 29th IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 613–622.
[41] S. He, X.-H. Sun, Y. Wang, A. Kougkas, and A. Haider, "A heterogeneity-aware region-level data layout scheme for hybrid parallel file systems," in *Proc. 44th Int. Conf. Parallel Process.*, 2015, pp. 340–349.
[42] S. He, Y. Wang, and X. Sun, "Boosting parallel file system performance via heterogeneity-aware selective data layout," *IEEE Trans. Parallel Distrib. Syst.*, vol. PP, no. 99, pp. 1–1, 2015.

Shuibing He received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, China, in 2009. He is now an assistant 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.

Yang Wang received the BSc degree in applied mathematics from the Ocean University of China, in 1989. He received the MSc and PhD degrees in computer science from the Carleton University, in 2001, and the University of Alberta, Canada, in 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.

Xian-He Sun received the BS degree in mathematics, in 1982, from the Beijing Normal University, China, and received the MS and PhD degrees in computer science, in 1987 and 1990, respectively from the 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, 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 an IEEE fellow.