S4D-Cache: Smart Selective SSD Cache for Parallel I/O Systems

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

***Abstract*—Parallel file systems (PFS) are widely-used in modern computing systems to mask the ever-increasing performance gap between computing and data access. PFSs favor large requests, and 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 a hybrid architecture named the Smart Selective SSD Cache (S4D-Cache), which employs a small set of SSD-based file servers as a selective cache of conventional HDD-based file servers. A novel scheme is introduced to identify performance-critical data, and conduct selective cache admission to fully utilize the hybrid architecture in terms of data-access parallelism and randomness. We have implemented an S4D-Cache under the MPI-IO and PVFS2 parallel file system. Our experiments show that S4D-Cache can significantly improve I/O throughput, and is a promising approach for parallel applications.**

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

I. Introduction

Many modern HPC applications are becoming increasingly data intensive. For example, the *astro* program in astronomy, generates tens of gigabytes of data in one run [1]. To meet the high I/O demands of these applications, HPC clusters rely on parallel I/O systems to provide data accesses. Typically, a parallel I/O system consists of several layers including applications, I/O middleware, parallel file systems (PFSs), and storage systems. In general, a parallel file system, such as PVFS [2], Lustre [3] and GPFS [4], will stripe file data across multiple file (I/O) servers and allows data requests to be served concurrently by multiple file servers. Thus, I/O system efficiency is significantly improved by exploiting parallelism when serving large I/O requests from a PFS.

While PFSs are an effective approach to cap the performance gap between the computing system and I/O system for large requests, they fail to perform well when serving clusters of small requests, especially random requests. In the meantime, hard disk drives (HDD), which are the dominant storage media deployed on current file servers, are notoriously slow in random data access due to the mechanical nature of disk head movements. Combining the difficulty of parallelism and slow access times, small random requests are easily the number one performance killer of PFSs.

To illustrate the performance degradation from this issue, we ran IOR [5] benchmark on a PVFS2 file system built on eight I/O servers (each server includes a single HDD). We limited the overall file size to 16GB, the number of processes to 16, and varied the request size from 4KB to 32MB. Each of the n MPI processes reads its own $1/n$ of the shared file, and continuously issues requests with sequential or random offsets. Figure 1 demonstrates the aggregated bandwidth for different request sizes during sequential and random I/O operations. The average bandwidth is reduced by more than half when small random accesses are conducted with different request size from 4KB to 32KB. For request size larger than 4MB, the random I/O performance is comparable to the sequential performance. These results confirm that small random access is a major performance impediment on parallel I/O systems.

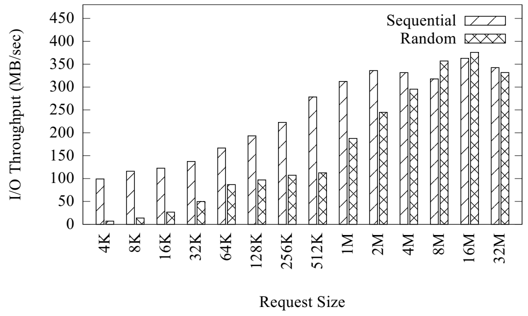


Fig. 1. I/O throughput for sequential and random reads

A number of approaches have been proposed in the I/O hierarchy to speed up the small random accesses over the past years. For example, I/O middleware approaches improve disk throughput by transforming a large number of small and non-contiguous requests into large contiguous requests [6], [7]. Memory caching strategies reduce the I/O latency by accessing more data from memory [8], [9]. I/O scheduling approaches reorganize the incoming I/O requests to create more sequential accesses in order to improve performance [10]. 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 drive (SSD).

Advanced storage devices, such as SSD, provide a possible hardware solution to improve small random access performance on PFSs. Currently, An SSD is commonly used as a

1063-6927/14 $31.00 © 2014 IEEE
DOI 10.1109/ICDCS.2014.59

cache [11]–[14] of HDD or as a hybrid storage [15], [16] on each file server. While this is straightforward to implement, it requires a large number of SSDs thus may be costly. Furthermore, since SSDs are deployed on each file server, the global utilization of SSDs becomes impossible which can be very useful to improve performance [17], [18].

In this paper, we propose the Smart Selective SSD Cache (S4D-Cache) architecture to combine the merits of SSDs with parallel file systems. The main idea is to employ a small set of SSD-based file servers as a selective cache for conventional HDD-based file servers. With a smart selective algorithm, S4D-Cache can significantly improve I/O throughput by buffering or caching large amounts of performance-critical data during both read or write requests.

Conventionally, a cache uses data locality principals to increase cache efficiency. However, a different scheme is used for the S4D-cache. The SSD-based cache in the S4D-Cache architecture is designed to utilize an SSD's ability to support small random data accesses. Therefore, the selection algorithm of S4D-Cache is derived from the randomness of data accesses, not the data access locality. Application-aware scheduling to utilize random access performance on SSDs and the parallelism on PFS is a key strength of S4D-Cache.

In summary, this study makes the following contributions.

- A novel system-level cache architecture is proposed to use SSDs as a selective cache on top of the traditional HDD-based file servers. This cache is pluggable and makes the employment of SSDs cost-effective.
- A cost model is introduced for parallel file requests, which can evaluate the access time of the request in parallel file systems built on different data storage media.
- A scheduling scheme which first identifies performance critical data via the cost model, and then uses a smart selective cache admission policy to take full advantage of the hybrid SSD and PFS architecture is introduced.
- A prototype of S4D-Cache is implemented and integrated under MPI-IO library on a computer cluster equipped with the PVFS2 parallel file system. This implementation is transparent to applications, and portable to many different parallel file systems. S4D-Cache is evaluated with representative benchmarks, including IOR, HPIO, and MPI-TILE-IO. Experimental results show that I/O throughput is significantly improved.

The rest of this paper is organized as follows. Section II discusses the related work. Section III describes the design of S4D-Cache and section IV presents the detailed implementation. Section V evaluates the performance of S4D-Cache with representative benchmarks. Finally, we conclude the paper in section VI.

II. Related work

This section focuses on previous work related to this study by looking at three separate aspects.

A. I/O Request Stream Optimization

To tackle the performance issue of small random requests, a lot of efforts have focused on I/O request stream reorganizations in the middleware layer. For multiple noncontiguous smaller requests, Data sieving [6] technique integrates them into a larger contiguous chunk including the additional data (hole) instead of accessing them separately. Datatype I/O [7] and List I/O [19] techniques allow users to merge multiple I/O requests with different patterns within a single I/O routine. Collective I/O [6] 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. S4D-Cache can use not only these techniques for its underlying parallel file systems but also utilize SSDs' characteristics.

B. Using System Memory as Cache

Traditionally the problem of small requests is addressed by using system main memory as cache. These cache schemes are deployed on both client side and server side in a parallel environment, including client-side file caching in GPFS [4] and Lustre [3], cooperative caching [20], active buffering [9] and collective caching [8].

In contrast to these memory-based methods, S4D-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, S4D-Cache has a totally different selection algorithm and runtime system design. The integration of memory cache and S4D-Cache will be an interesting topic for future study.

C. SSD-based Storage System

Using SSDs as a cache of traditional HDDs is a widely used strategy in I/O systems, such as FlashCache [12], Conquest [21], SieveStore [14], iTransformer [13], and iBridge [11]. Liu et al. simulates a system using SSD storage on I/O nodes as buffers to handle burst I/O requests [22]. Tiered checkpointing redirects all write data to the RAM disks or SSDs in the computing nodes [23]. SSD-based hybrid storage is another popular method to make full use of SSD. This method integrates an SSD and a hard disk as one block device [24], [25]. I-CASH is a new hybrid storage architecture based on data-delta pairs to improve I/O performance for I/O-intensive workloads [16]. Hystor identifies critical data blocks with strong temporal locality and redirects them to SSD for fast future accesses [15].

These approaches succeed in exploiting data access information within single file server or computing node. But unlike this work, S4D-Cache leverages the global data access information in parallel environment to improve I/O performance. Our previous work CARL similarly uses the global data information and SSDs to boost performance [26]. However, the SSD-based servers are used as persistent storage instead of cache. With a small set of SSD-based file servers and the selective

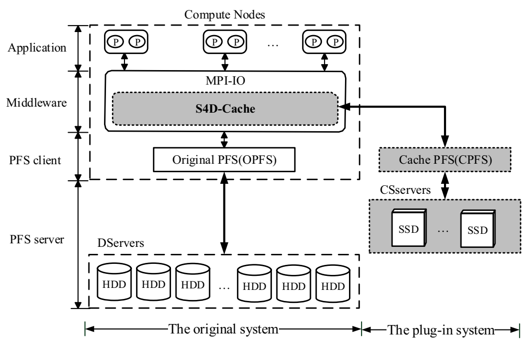


Fig. 2. The S4D-Cache architecture

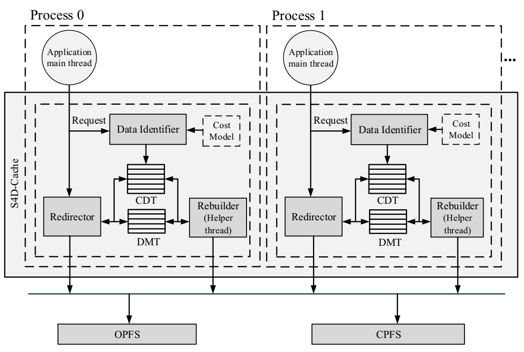


Fig. 3. Software structure of S4D-Cache

cache policy, S4D-Cache provides a feasible and cost-effective solution for large-scale data intensive applications.

III. Design of S4D-Cache

S4D-Cache aims to use SSD-based file servers to cache small random accesses in parallel I/O system with non-uniform workloads. By exploiting SSD's strong performance advantage for small random request, the I/O system performance can be significantly improved.

A. Architecture Overview

Figure 2 shows the high performance computer systems for which S4D-Cache is designed. S4D-Cache acts as an augmented module to MPI-IO library [27]. In these systems, besides the traditional HDD-based file servers (DServers), there are a small number of SSD-based File servers (CServers). DServers are accessed by the original parallel file system (OPFS); CServers act as a fractional cache of DServers and are accessed by the cache parallel file system (CPFS). When application processes pass their I/O requests to MPI-IO, S4D-Cache intercepts all the requests and choose the proper servers to serve them.

Positioning S4D-Cache at the middleware layer is ideal for several reasons. First, key global data access information, such as file-level, process-level, and MPI Rank-level attributes are accessible. Second, the middleware layer is independent of the file system, allowing the solution to support multiple file systems, such as PVFS [2], Lustre [3], and GPFS [4]. Third, the plug-in design is transparent to applications, therefore user programs do not require modifications to utilize the increased performance. Finally, because only a small cluster of SSDs are deployed into the system, the design is flexible and highly cost-effective.

S4D-Cache consists of three key software components: *Data Identifier*, *Redirector* and *Rebuilder*, as shown in Figure 3. *Data Identifier* intercepts every file request issued to DServers, and identifies requests for performance-critical data using a data access cost model. *Redirector* redirects the selected requests to the high-performance CServers. While selected write requests and cached read requests are redirected to CServers, other write requests and missed read requests are directed to the traditional DServers. *Rebuilder* is responsible for flushing the selected write data back to DServers, and fetching the selected read data to CServers.

B. The Data Access Cost Model

TABLE I
Parameters (short in Pars) in cost analysis model.

Pars	Description
M	Number of HDD file servers
N	Number of SSD file servers ($N < M$)
str	Stripe size of parallel file system
d	Logical address distance between r_i and r_{i-1}
f	File offset of request r_i
r	Data size of request r_i
R	Average rotation delay for HDD
S	Maximum seek time for HDD
β_D	Cost of access one unit of data for HDD
β_C	Cost of access one unit of data for SSD

Because each CServer is of relatively limited size, S4D-Cache only caches performance-critical data. Thus, the potential performance benefit of redirecting a request to CServers 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 parallel file systems, and the corresponding parameters are listed in Table I.

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

$$T_D = T_s + T_t \tag{1}$$

T_s is the startup time, including disk seek and rotation delay. T_t is the data transfer time spent on actual data movement. Let α denote the startup time in each server, then α is usually a random variable. Assume α follows uniform distribution on [a, b], then the probability function of α is

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

Here $a = F(d) + R, b = S + R$. The request distance d can be a metric to measure the randomness of a request, and F is a function for converting d to seek time. We use the approach described in [28] to derive this function from an offline profiling of the HDD storage.

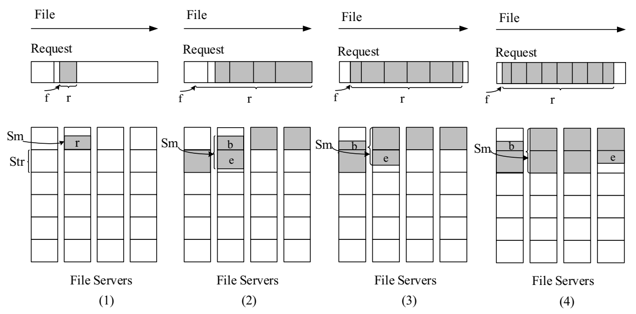


Fig. 4. Four cases where a file request involves a different number of sub-requests.

For a parallel request req, it may involve multiple sub-requests on m file servers, and the overall startup time of req is determined by the maximum of all sub-requests. Let α_1, α_2, $\cdots$, α_m be the startup time of the m file servers, $X = max(\alpha_1, \alpha_2, \cdots, \alpha_m)$, then the probability density function of X is

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

Hence, the expectation of the maximum startup time

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

On the other hand, the data transfer time T_t of request req should be the maximum of all m file servers, which is proportional to the data size in each file server. Let $s(i)$ is the size of the sub-request on file server i ($1 \leq i \leq m$), and $s_m = max\{s(1), s(2), ..., s(m)\}$, then

$$T_t = s_m * \beta_D \tag{5}$$

Assume the parallel file is placed on DServers with a fixed-size stripe in a round-robin way, then for a given request 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 $E = \lfloor \frac{f+r}{str} \rfloor$. Hence, the number of the involved file servers is

$$m = \begin{cases} E - B + 1, & E - B + 1 < M \\ M, & otherwise \end{cases} \tag{6}$$

Accordingly, the size of the beginning fragment can be calculated as $b = str - f\%str$, and the size of the ending fragment is $e = (f + r)\%str$. For a parallel I/O request, it will be served by multiple file servers concurrently. Figure 4 shows an example of the possible sub-request layouts of req. Let $\triangle = E - B$, then $\triangle \geqslant 0$, and s_m can be calculated as table II. Based on Equation 6 and table II, T_D of each file request in Equation 1 can be obtained.

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

TABLE II
MAXIMUM SIZE OF SUB-REQUEST IN DIFFERENT FILE ACCESSES CASES.

Case	Maximum size of sub-request (s_m)	Conditions
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

$$T_C = S_n * \beta_C \tag{7}$$

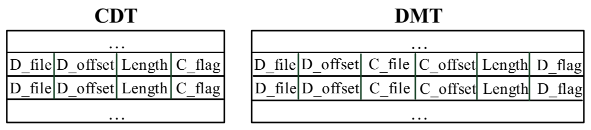
Then the performance benefit of serving a request issued to DServers if it were served by CServers can be calculated as following:

$$B = T_D - T_C \tag{8}$$

C. Critical Data Identification

With the proposed data access cost model, *Data Identifier* is able to obtain the performance benefit (*B*) for each incoming request. By examining the above equations, it can be noted that small random requests lead to more benefit from CServers, because single CServer has performance advantage in serving them. However, large continues requests will get less or even no benefit because DServers have higher parallelism due to more file servers. A positive *B* means that serving the request at CServers will reduce the I/O access time, i.e., increase the parallel I/O system performance. In such a case, since the space of CServers is limited, the request should be served at CServers. Otherwise, serving the request at DServers helps improve the I/O performance and there is no need to serve it at CServers.

When the benefit *B* of a request is larger than zero, *Data Identifier* regards the requested data as performance-critical data, and records it to the critical data table (CDT). As shown in Figure 5, each entry in CDT consists of four variables, D_file, D_offset, Length, and C_flag. They indicate the file name in DServers, the data offset in the file, the data length, and whether the data needs to be cached in CServers, respectively. Based on the CDT, critical read/write data can be identified and redirected to the high-performance CServers.


CDT

…			
D_file	D_offset	Length	C_flag
D_file	D_offset	Length	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. 5. The data structure of CDT and DMT

D. Cache Metadata Management

S4D-Cache creates a correlating cache file for each origial file and uses Data Mapping table (DMT) to keep track of data information that has been cached in CServers. As shown in Figure 5, 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 CServers contains data that requires to be copied back to DServers. The DMT is updated each time a data location has changed. By maintaining the DMT, *Redirector* can continuously track the most up-to-date location of the data, which ensures data consistency between DServers and CServers.

In memory, the table is organized as a hash table to speedup lookups, which only incur minimal overhead with several memory accesses. Since only remapped data needs to be tracked in the mapping table, 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 in CServers. Changes to the mapping table are synchronously written to the storage in order to survive power failures.

In parallel I/O environment, multiple processes possibly access DMT concurrently. In order to keep metadata consistency, 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 the mechanism in Berkeley DB to perform metadata operations and address lock contentions. Techniques, similar to the distributed cache meta data [8], can also be applied to distribute meta data among the application processes, so that the communication contention for accessing metadata can be minimized.

E. Selective Caching Scheme

Redirector is a core module in S4D-Cache, it selectively caches data based on four factors: (1) the mapping entry in DMT, indicating if the request can be served by CServers, (2) the entry in CDT, indicating if the missed request should be admitted in CServers, (3) type of I/O request (read or write), and (4) the available space in CServers.

Upon each I/O request, *Redirector* looks up the DMT and checks if the request hits CServers or not. If so, *Redirector* directly serves the request with the data in CServers. Otherwise, *Redirector* handles the request obeying a selective cache policy.

Algorithm 1 shows the work-flow of *Redirector* for each I/O request. The algorithm attempts to utilize CServers whenever possible. For write requests, CServers are regarded as a write buffer. If there is a sufficient space in CServers (line 5 and 10) or the request is already mapped (line 22), the request will be absorbed by CServers. In order to reduce data migration overhead, the algorithm first looks for free space in CServers when allocating an available space for a write request. If free space cannot be found, a clean space will be the candidate based on a LRU policy. For read requests, *Redirector* uses CServers as a caching area. When the required data misses,

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 in CServers
 5:       if free space is found then
 6:         add new entry in DMT (mark dirty)
 7:         change the $req$ location as the DMT entry
 8:       else
 9:         find clean space in CServers
10:         if clean space is found then
11:           change the entry in DMT (mark dirty)
12:           change the $req$ location as the DMT entry
13:         end if
14:       end if
15:     end if
16:   else
17:     if $req$ is in CDT then
18:       set the C_flag of the entry in CDT
19:     end if
20:   end if
21: else
22:   change the $req$ location as the DMT entry
23: end if
24: send request $req$
```

the request is cached in a "lazy" way. This means that Redirector marks the C_flag in the corresponding entry of CDT (line 18), which indicates to *Rebuilder* that an actual data movement should be performed 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 the CDT (line 3 and 17), to maximize use of CServers space.

F. Data Reorganization

Rebuilder plays the role of freeing CServers space for future use. It is triggered periodically, and performs two kinds of operations. 1) It writes dirty data back to DServers, 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 DServers into CServers 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.

IV. Implementation

We have implemented the S4D-Cache selective scheme and its runtime system under MPICH2 [29]. The primary and challenging parts are explained below.

A. Cache Metadata Mapping Table

Both *Redirector* and *Rebuilder* need to get application data access information from the DMT. The DMT is a key structure to save the mapping relation between the data cached in CServers and DServers.

We use Berkeley DB to implement the DMT table. DMT is a database file which has a standalone space in CServers. The Berkeley DB is configured as a hash table, and each record is a key-value pair. We generate a mapID by encoding the following information: application name, number of process, rank of the process, and the original file name. 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 Figure 5. By leveraging the light-weighted DB, the lock contention is addressed and metadata operations are performed efficiently. We also use a list to maintain the most frequently accessed mapping entries which further reduces the in-memory mapping table size.

B. I/O Redirection Module in MPI-IO

The I/O redirection module redirects data accesses on the original files to the cache file. 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 a corresponding cache file.

MPI_File_read: For each I/O read, this method first uses the input parameters to calculate the performance using Equation 8. Next, the corresponding entry is added to the CDT table if the request is a critical request and not in the CDT; then the method checks whether the opened cache file contains the requested content by looking up the DMT table. If this condition 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. If the access data belongs to the CDT, the method sets the C_flag of the entry in the CDT table, which will be later used by *Rebuilder*.

MPI_File_write: For each I/O write, this module uses the input parameters to calculate the performance benefit and adds the corresponding entry to the CDT table as necessary. Then, the module checks whether the opened cache file contains the requested content by initiating a lookup in the DMT table. If this is true, 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 the CDT. If so, the method tries to allocate available space in the cache file 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: It closes the opened cache file.

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

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.

C. 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.

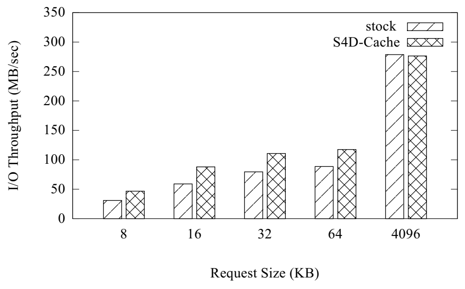
V. Performance Evaluation

In this section, the performance of prototype implementation of S4D-Cache is evaluated through extensive experiments.

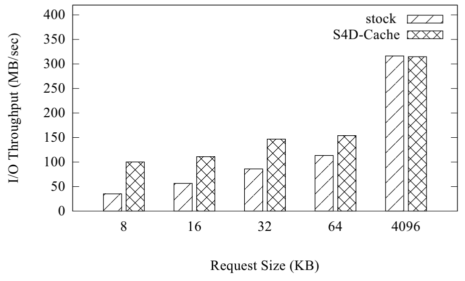
A. Experimental Setup

The experiments were conducted on a 65-node SUN Fire Linux cluster. Each computing node has two AMD Opteron processors, 8GB memory and a 250GB HDD (SEAGATE ST32502NSSUN250G). The operating system is Ubuntu 9.04 and the parallel file system is PVFS2 version 2.8.2. All nodes are equipped with Gigabit Ethernet interconnection, and eight nodes are equipped with an additional PCI-E X4 100GB 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 S4D-Cache.

Among the available nodes, 32 nodes are used as computing nodes, eight are DServers, and four are CServers. Each DServer uses the HDD as storage and each SServer uses the SSD. DServers and CServers are separately accessed with their PVFS2 parallel file system. MPICH2 [29] compiled with ROMIO is used to generate the executable. When S4D-Cache is enabled, the cache capacity is set to 20% of the application's data size. S4D-Cache does not benefit read performance if the requested data have not been cached in CServers. However, many MPI programs are executed several times and present consistent data access patterns [17], [30]. The critical data identified and cached by S4D-cache in the first run can improve read performance in the later runs. Therefore, the read performance improvement of S4D-Cache for the program with a second run is shown in this paper. In order to show the effectiveness of SD-Cache, the benchmarks of IOR [5], HPIO [31], and MPI-Tile-IO [32] are used to evaluate the performance.



(a) Throughput for write



(b) Throughput for read

Fig. 6. I/O throughputs of IOR with varied request sizes.

B. The IOR Benchmark

IOR is a parallel file system benchmark developed at Lawrence Livermore National Laboratory [5]. It provides three APIs: MPI-IO, POSIX, and HDF5; however, only the MPI-IO is used in this benchmark. To simulate different data access patterns at different moments, 10 instances of IOR are created one by one with different parameters. Among these instances, six issue sequential I/O requests and the remaining send random I/O requests. In each instance, the test performs write and read operations to a shared 2GB file. During these benchmarks, 32 processes are used and the request size is kept to 16KB unless otherwise specified.

1) Varying Request Sizes: While varying the request size of IOR from 8KB to 4096KB, the overall I/O performance is measured. As shown in Figure 6(a), S4D-Cache can improve the overall write throughput by 51.3%, 49.1%, 39.2% and 32.5% over the stock I/O system with the request size of 8KB, 16KB, 32KB and 64KB respectively. For request size of 4096KB, S4D-Cache nearly has the same I/O throughput as the stock I/O system. With smaller request sizes, the I/O throughput improvement is more significant because CServers can lead to more benefits for them. For large continuous requests, as DServers has higher parallelism and the performance gap between CServer and DServer is reduced, placing them on CServers incurs less or no performance benefits. Thus, S4D-Cache can bring less performance improvement.

To better understand the reasons for the performance improvement, the accessed addresses of requests on DServers and CServers are tracked using IOSIG, an I/O pattern analysis tool developed in our previous work [33]. Table III shows the request distribution at DServers and CServers during the five-second period of IOR execution from the 50th second with write request sizes of 16KB and 4096KB. For 16KB requests, most of the requests are redirected to CServers, and DServers mostly sees sequential requests. When the request size is 4096KB, because serving them from DServer will lead to higher performance, all the requests are dispatched to CServers. This result shows S4D-Cache can effectively identify the performance-critical data and redirect to them to CServers to improve performance.

TABLE III
REQUEST DISTRIBUTION.

Request Size	DServers (%)	CServers (%)
16KB	16.3	83.7
4096KB	100.0	0.0

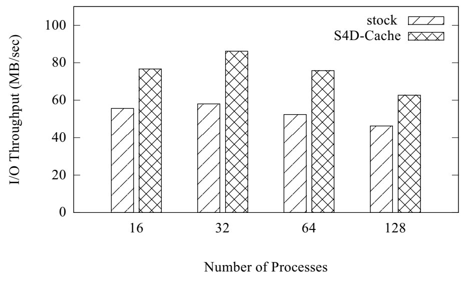
The read test yields similar result, as shown in Figure 6(b). S4D-Cache can increases the throughput by up to 184.1% with the request size of 8KB. Compared to the write test, S4D-Cache has a larger improvement in read because the SSD performs better for reads than for writes.

2) Varying Number of Processes: Each instance of IOR benchmark was run with 16, 32, 64, and 128 processes. Different processes access various regions of the original file so that no process' data co-locates with any other's data. Figure 7(a) gives the results of this test for write. Similar to the previous test, S4D-Cache improves the overall I/O bandwidth by 35.4% to 49.5%. With the number of processes increasing, IOR's bandwidth gets lower because each file server needs to serve more processes' requests and the competition among processes gets more severe. This result also shows S4D-Cache has a good scalability in terms of the number of processes. The performance trend is similar for read requests, as shown in Figure 7(b).

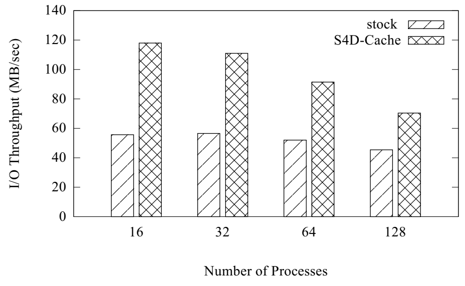
3) Varying SSD Cache Capacities: In general, the capacity of CServers is much smaller than that of DServers and could be even smaller than the I/O working set size for the application. According to the algorithm, S4D-Cache could elastically replace the cached data to increase the utilization of the SSD space. Table IV shows the write throughputs when the SSD cache capacity is varied from 0GB to 6GB. Here 0GB means that S4D-Cache is disabled. It is observed that I/O throughput improves by increasing the capacity of CServers, which is because more random I/O requests can benefit from CServers. However, when most random requests are already cached (the capacity is above 4GB), continuously enlarging CServers will only bring limited performance improvement.

4) Varying Numbers of SSD file servers: Finally, the number of the file servers in CServers is varied while maintaining the same available cache space and I/O access patterns. IOR was benchmarked with different number of SSD file servers from zero to six. (0 means the stock I/O system is used.)

Figure 8(a) shows the results for write operations. The overall write bandwidth is improved by 20.7% to 60.1%. With the number of CServers increasing, the I/O bandwidth improves because CServers can serve the redirected requests

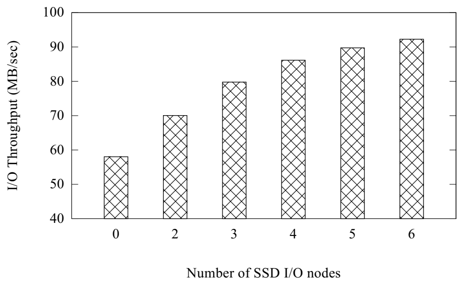


(a) Throughput for write

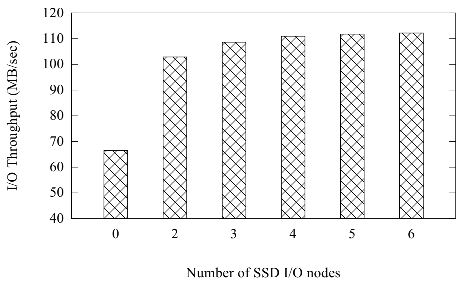


(b) Throughput for read

Fig. 7. I/O throughputs of IOR with varied numbers of processes.



(a) Throughput for write



(b) Throughput for read

Fig. 8. I/O throughputs for the IOR benchmark with varied numbers of CServers.

TABLE IV
I/O THROUGHPUTS OF IOR WITH VARIED SSD CACHE CAPACITIES.

SSD Capacity	Throughput (MB/s)	Speedup (%)
0GB	58.03	0
2GB	69.34	19.5
4GB	86.15	48.4
6GB	90.89	56.6

with better random performance. However, the improvement reduces when continuously adding more nodes to CServers. More specially, the I/O performance only slightly improves when the number of file servers is above four. This is because only a portion of the I/O workload is random 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 the SSDs. For reads, the I/O throughput is higher writes, due to the better random read performance of SSD, but it also has a plateau, as shown in Figure 8(b).

C. The HPIO Benchmark

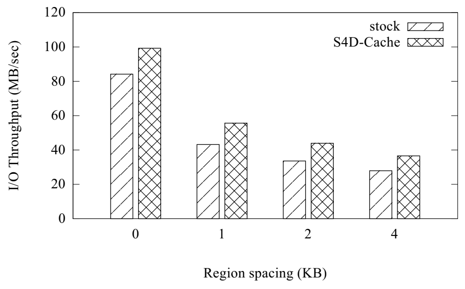
HPIO is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate I/O [31]. This benchmark can generate various data access patterns by changing three parameters: region count, region spacing, and region size. The region spacing is used to generate noncontiguous data access patterns. In our experiment, the number of process is set to 16 processes; the region count is set to 4096; the region size is set to 8KB; and region spacing is varied from 0KB to 4KB(0KB indicates sequential access).

As shown in Figure 9(a), S4D-Cache can increase the I/O throughput by 18%, 28%, 30%, and 33% respectively. It means that S4D-Cache is effective with respect to HPIO benchmark. 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 S4D-Cache; 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 S4D-Cache. For read operations, the performance has similar trend as presented in Figure 9(b).

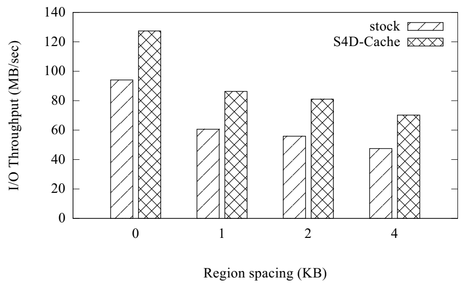
D. The MPI-Tile-IO Benchmark

MPI-Tile-IO is a test application from the Parallel I/O Benchmarking Consortium [32]. 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 in the X and Y directions are set to 10 and 10, the size of each element is set to 32KB, and the number of processes is varied between 100 and 400.

Figure 10 shows the aggregated I/O throughputs. The aggregated bandwidth increases by 21% to 33% for writes, and 18% to 31% 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 test. 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 S4D-Cache brings additional benefits when data requests are more random in nature.

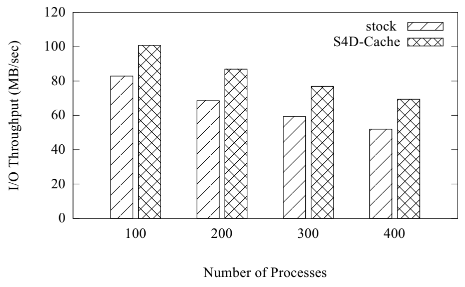


(a) Throughput for write

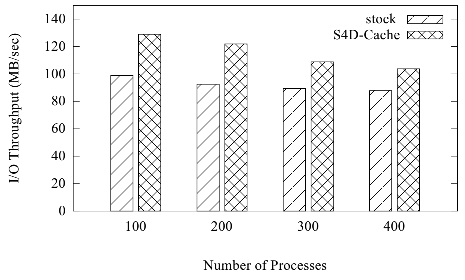


(b) Throughput for read

Fig. 9. I/O throughputs of HPIO with varied region spacings.



(a) Throughput for write



(b) Throughput for read

Fig. 10. I/O throughputs of MPI-Tile-IO with varied number of processes.

E. System Overhead

1) Metadata Space Overhead: To track data cached in CServers and maintain data consistency, S4D-Cache uses a file in CServers to store the DMT Table. This insures additional storage space overhead. The amount of space consumed is maximized when all the request sizes are 4KB.

Assuming the available storage space of CServers is S(GB); and each entry in our implementation occupies $6 * 4$B, the maximum number of records in DMT is $S/4 * 10^6$. Thus, the metadata space overhead is 0.6%, which is negligible.

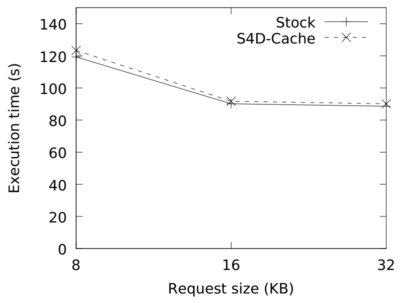


Fig. 11. Performance overhead result.

2) Performance Overhead: As shown in Figure 3, S4D-Cache has some additional modules which may generate overhead. S4D-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, the I/O identifier module needs to initialize the DMT table in memory, and decide whether to create a new cache file in CServers.

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

The two overheads mentioned above are very small compared with I/O operations overheads. In order to show the overhead is negligible, IOR is benchmarked with 32 processes, a request size which varies from 8KB to 32KB, and each process writes a shared 10GB file in a random pattern where all the requests intentionally miss the CServers. This causes the Redirector to redirect all requests to DServers. Figure 11 shows these results. As expected, the overhead is almost unobservable.

VI. Conclusions

In this study, we have introduced the Smart Selective SSD Cache (S4D-Cache) system to combine SSD devices with parallel I/O systems. S4D-Cache deploys a small set of SSD file servers to cache selected performance-critical requests on top of a large set of conventional parallel HDD file servers. The S4D-Cache design has several merits: (1) it is cost-effective because only a small set of SSD are deployed; (2) it is smart in fully utilizing the value of parallelism and the newly emerged storage media, SSDs; (3) its plug-in design is transparent to applications; therefore, user programs do not need to be modified; (4) its design is general and can be applied to different parallel file systems.

The data selective functionality of S4D-Cache is supported by three different system components: *Data identifier*, *Redi-*

rector and *Rebuilder*. *Data identifier* identifies performance-critical data from I/O request streams. *Redirector* makes the data selection decision of the identified critical data to fully utilize SSDs based on the access cost analysis. *Rebuilder* reorganizes the content of the SSD cache space and conducts data movements based on the changes of I/O workload.

The S4D-Cache design is implemented under MPICH2 I/O and the PVFS2 parallel file system environment. Its performance is evaluated with different benchmarks, namely IOR, HPIO, and MPI-TILE-IO, on a SSD-equipped computer cluster. Experimental results show that S4D-Cache is feasible and effective in improving parallel I/O performance.

Acknowledgment

The authors are thankful to their shepherd André Brinkmann and the anonymous reviewers for their helpful comments in reviewing this paper. This research was supported in part by National Science Foundation under NSF grant CNS-0751200, CCF-0937877 and CNS-1162540.

References

[1] M. Kandemir, S. W. Son, and M. Karakoy, “Improving I/O performance of Applications through Compiler-DirectedCode Restructuring,” in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 159–174.

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

[6] 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.

[7] 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.

[8] W.-k. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tidemad, “Collective Caching: Application-Aware Client-Side File Caching,” in *Proceedings of the 14th IEEE International Symposium on High PerformanceDistributed Computing*, 2005, pp. 81–90.

[9] X. Ma, M. Winslett, J. Lee, and S. Yu, “Improving MPI-IO Output Performance with Active Buffering plus Threads,” in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[10] Y. Xu and S. Jiang, “A Scheduling Framework that Makes any Disk Schedulers Non-work-conserving Solely Based on Request Characteristics,” in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

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

[12] M.Srinivasan and P. Saab, “Flashcache: A General Purpose Writeback Block Cache for Linux,” 2013. [Online]. Available: https://github.com/facebook/flashcache

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008, pp. 1–12.

[19] 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.

[20] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, “Cooperative Caching: Using Remote Client Memory to Improve File System Performance,” 1994.

[21] 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 *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002, pp. 15–28.

[22] 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 *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, 2012, pp. 1–11.

[23] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, “Design, modeling, and evaluation of a scalable multi-level checkpointing system,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[24] 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.

[25] 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.

[26] 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.

[27] R. Thakur, W. Gropp, and E. Lusk, “On Implementing MPI-IO Portably and with High Performance,” in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999, pp. 23–32.

[28] H. Huang, W. Hung, and K. G. Shin, “FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption,” in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2005, pp. 263–276.

[29] 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

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

[31] 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.

[32] “MPI-Tile-IO Benchmark.” [Online]. Available: http://www.mcs.anl.gov/research/projects/pio-benchmark/

[33] 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.