ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage

Bin Dong, Teng Wang, Houjun Tang, Quincey Koziol, Suren Byna, Kesheng Wu

Lawrence Berkeley National Laboratory, Berkeley, CA 94720
Email: {dbin, tengwang, htang4, koziol, sbyna, kwu}@lbl.gov

***Abstract*—Scientific data analysis typically involves reading massive amounts of data generated by simulations, experiments, and observations. A significant bottleneck in this process is reading such data because the data files are stored on the rotating disks. Recent supercomputing systems are adding non-volatile storage layers to fill the performance gap between fast main memory and the slow disk-based storage. Software libraries for managing this hierarchy not only need to read data efficiently, but also reduce user-involvement for cross-layer data movement. As the scientific data is usually organized as arrays, these libraries also need to support array data access patterns over hierarchical storage systems. Existing software tools manage individual storage layers separately, and require significant manual work to move data among the layers. In this paper, we introduce a new array caching in hierarchical storage (ARCHIE) to accelerate array data analyses in a seamless fashion. ARCHIE evaluates array access patterns and prefetches data with array semantics between storage layers. On a production supercomputing system, our evaluation shows that ARCHIE outperforms state-of-the-art file systems, i.e., Lustre and DataWarp, by up to $5.8\times$ in accessing data by scientific analysis applications.**

***Index Terms*—Caching and prefetching, HDF5, Data Elevator, hierarchical storage, burst buffers**

I. Introduction

Scientific data analysis applications running on high-performance computing (HPC) systems often spend significant amount of time in reading data [14], see Fig. 1. This is because the underlying data files are stored in disk-based parallel file systems, which require considerable amount of time to the terabytes and petabytes required for the analysis operations. In Fig. 1, we show the percentage of time for reading data by three scientific data analysis codes that were run on a Cray XC40 system, called Cori, at the National Energy Research Scientific Computing Center (NERSC). We observe that the data access time ranges between 24% and 86% of the total execution time.

While recent advances in HPC storage systems are adding multiple levels of storage hierarchy, traditional file systems are not equipped with supporting hierarchy of storage layers. To handle the bursty nature of generating and writing data, exascale computing designs are including node-local, non-volatile storage, and shared burst buffers [6]. These layers act as faster caching layers to store the data temporarily. However, traditional file systems are designed for managing a single layer of storage devices, such as arrays of disks or solid-state drives (SSD) [10], [22]. While there are solutions for moving

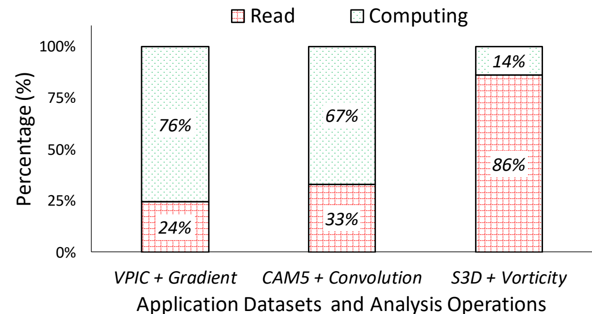


Fig. 1: From a profile of data access time and computation time for three scientific data analysis codes, we see significant portions of overall time is spent on reading data. ('VPIC + Gradient' is a gradient calculation of a plasma physics simulation (VPIC) dataset, 'CAM5-Convolution' is a Convolution operation for a climate simulation dataset, and 'S3D + vorticity' is vorticity calculation of a combustion dataset.)

the data between fast non-volatile storage and slow disk-based storage, solutions to accelerate reading data by prefetching into the intermediate storage layers are still required. While a few systems support prefetching the entire files, they often require user involvement. For example, Cray DataWarp [6] provides tools for users to manually or programatically move data to a faster shared burst buffer. Similarly, software such as DDN IME [7] and DAOS [17] also need storage infrastructure changes or user involvement in data movement.

Moreover, solutions using array semantics and prefetching data at smaller granularity than the entire file are unavailable. Scientific data analysis frameworks such as TensorFlow [1] and ArrayUDF [9] work on large multi-dimensional arrays, e.g., 2D images. Incorporating array semantics into hierarchical storage optimizations can help to further reduce data reading cost in data analysis tasks. Meanwhile, these data analysis frameworks are mostly designed to operate on a single storage layer. Providing a transparent way to use hierarchical storage for these systems is a critical task.

With the goal of providing a transparent and efficient data prefetching solution using array semantics in a hierarchical storage subsystem, we propose a Array Caching in HIErarchical storage system (ARCHIE). This system is

parallel by design and runs concurrently with data analysis applications to prefetch array data sets. ARCHIE provides an "array semantics"-aware prefetching function to move array data from slow storage devices into faster ones that are closer to analysis applications before the application issues the data read calls. This prefetching system reduces I/O overhead in most array-based data analysis applications. The technical contributions of this effort are:

- Development of a new array cache management system for hierarchical storage to move large array data efficiently across heterogeneous devices.
- Design of a parallel prefetching method for large arrays to improve data read performance in analysis tasks. Our method also augments prefetched data with ghost zones, a typical requirement in array-based data analysis.
- Development of a fault-tolerance mechanism for the caching system to recover from errors automatically that may crash an application or the ARCHIE service.
- Implementation of ARCHIE using the HDF5 Virtual Object Layer (VOL) [4] to provide caching and prefetching capabilities without requiring source code modifications.

We demonstrate the effectiveness of ARCHIE with three scientific data analysis codes, including convolution neural network [21], gradient computing [9], and vorticity analysis [5]. We ran our experiments on the Cori HPC system at NERSC that is equipped with a hierarchical storage system and thousands of computing nodes. The results show that ARCHIE is up to $6X$ faster than DataWarp [6], the state-of-the-art storage system for managing burst buffers.

II. Background

A. Hierarchical Storage Systems

Several pre-exascale and exascale computing systems are being deployed or designed with multiple levels of storage device layers to reduce the latency gap between main memory and disk-based storage. Adding an extra layer of non-volatile storage, such as SSDs, has been adopted on systems that have been deployed lately. There are two methods of integrating SSDs into a supercomputing system, 1) mounting SSDs on each computing node and 2) mounting SSDs on dedicated nodes remotely. An example of the former case is the Theta system at the Argonne Leadership Computing Facility (ALCF), where each compute node has a 128GB SSD[1]. An example of the latter case is the Cray XC40 system at NERSC, called Cori, which contains 144 Linux nodes (called burst buffer servers) with SSDs providing a single namespace managed by Cray DataWarp [2].

Various solutions have been developed to manage these storage devices as a unified namespace. For instance, BurstFS [28] presents an ephemeral namespace using node-local storage devices. As mentioned above, DataWarp [6] uses XFS to unify storage on burst buffer servers and provides the POSIX-IO interface for users to access its data. DataWarp also provides tools for users to move data in and out of the unified SSD space, such as *stage_in* and *stage_out*. Both BurstFS and DataWarp distribute large files to multiple SSD devices for parallel access. These solutions assume a disk-based file system (such as Lustre [10]) for long-term and persistent data storage. Therefore, users need to deal with hierarchical storage system with both SSD space and disk space.

The main limitation of both DataWarp and BurstFS is their lack of efficient functions to support reading data [8]. For instance, in DataWarp, reading data for analysis relies on users to *stage_in* entire data files from a file system into a shared SSD-based burst buffer, as shown in Fig. 2. Analysis applications can read the data from the SSD. This method increases the **end-to-end time**[3] for data analysis applications. As described in following sections, a large data analysis may split data into smaller subsets and read these subsets for analyzing sequentially. Thus, it is possible to have advanced storage optimizations to overlap data analysis and reading the subsets that would be required in future, without delaying applications' start until the completion of *staging in* files. As discussed in the following paragraphs, scientific data analysis typically work on array data. Incorporating array semantics into software for hierarchical storage system can further allow such optimization to reduce I/O cost of analysis applications.

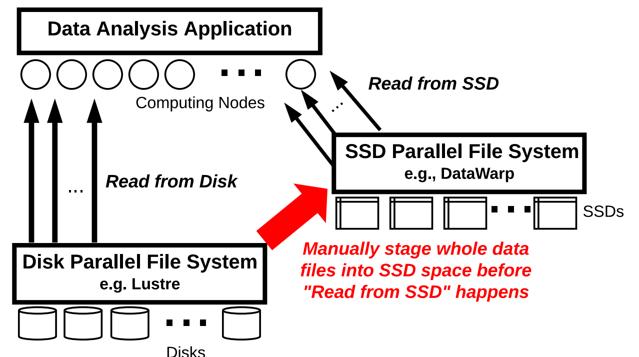


Fig. 2: An overview of hierarchical storage, with a disk-based parallel file system (e.g., Lustre) and an SSD-based file system (e.g., DataWarp). Data files to be analyzed are usually stored in Lustre for long-term persistence. To use the faster SSD-based storage, currently entire data files need to be *manually* staged in on DataWarp before applications can read them. Obviously, this approach exacerbates space utilization and extends the end-to-end time for applications.

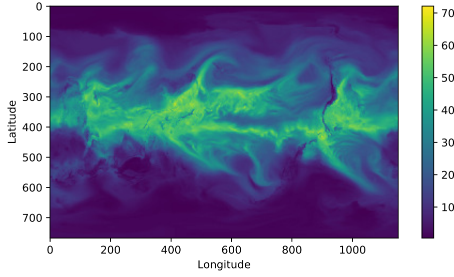
B. Representation of an Array

An *Array* is a basic data structure to store data elements with the same data type. An Array can be viewed as a mapping function from a Cartesian product to a set of attribute spaces: $[D_0, D_1, \ldots, D_{d-1}] \mapsto < A_0, A_1, \ldots, A_{m-1} >$, where D_i is a set of dimensions and A_i is a set of attributes.

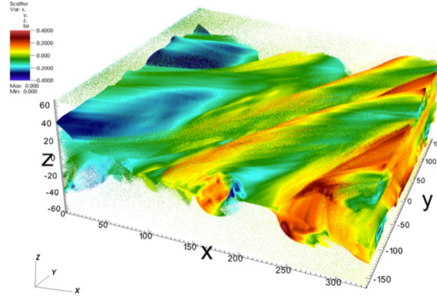
[1]https://www.alcf.anl.gov/theta

[2]http://www.nersc.gov/users/computational-systems/cori/burst-buffer/

[3]The end-to-end time refers to the time taken by users to move the entire data files from Lustre into DataWarp and then read them from DataWarp into application memory buffers for analysis.



(a) CAM5



(b) VPIC

Fig. 3: Example array datasets in scientific applications: a) A 2D array represents the total (vertically integrated) precipitable water (kg/m2) generated by CAM5 [32]. b) Mapped 3D magnetic field data in the z dimension (bz) for all highly energetic particles ($E > 1.5$) in physical space (x, y, and z dimensions) (Image produced by O. Rübel [3]).

D_i is a continuous range of integer. Arrays are found in many scientific applications. We briefly discuss two analysis operations of two science use cases with large array data to motivate optimizations of reading data:

- **Convolutional Neural Networks (CNN) on climate data**: The Community Atmospheric Model version 5 (CAM5) [32] is a global atmospheric model that uses the finite volume dynamical core on a latitude—longitude mesh. A typical resolution is 0.23° (latitude) by 0.31° (longitude), giving a 768 by 1152 array. Figure 3(a) presents one attribute, named the total precipitable water, on the latitude-longitude mesh. A recent study [21] uses a 3D CNN with a bounding-box regression analysis to detect and identify extreme weather events, e.g., tropical depressions and tropical cyclones. The main computation task in CNNs is the convolution operation, which is defined as:

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n-m) \quad (1)$$

  where f, g are two complex-valued functions on the set Z of integers. In 2D image analysis, $f(m)$ represents a set of neighborhood pixels and $g(n-m)$ is the filter/kernel function (i.e., weight parameter) on each pixel.
- **Gradient computing on plasma physics data**. The Vector Particle-in-Cell (VPIC) [3] is used to understand plasma dynamics in strong magnetic field systems. The core algorithm solves for the electric and magnetic fields on a discrete "Yee" mesh using a finite-difference-time-domain method. The magnetic field data produced by VPIC is usually a 3D dataset, as shown in Fig. 3b. A key operation in analyzing VPIC data is finding the gradient of the magnetic field, which is essential for understanding particle acceleration. On a meshed 3D field, the gradient can be computed with a Laplacian as :

$$\text{grad} f(x,y,z) = \nabla f(x,y,z) \quad (2)$$

  where f represents magnetic value and ∇ denotes the Laplace operator.

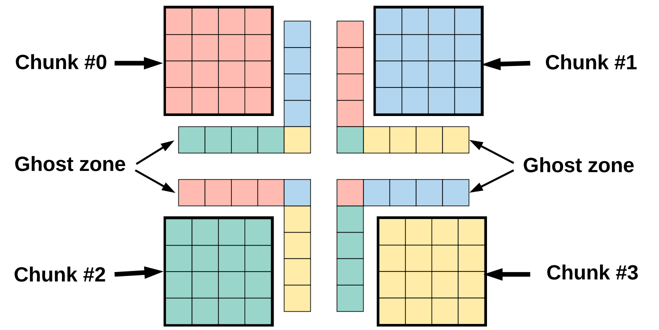


Fig. 4: An example $[8, 8]$ 2D array with chunking and ghost zone. The size for chunking is $[4, 4]$ and for ghost zone is $[1, 1]$. This pattern widely exists in data analysis. For example, in the CNN analysis on CAM5 data. A chunk may contain all cells of a convolution filter. Since the convolution filter may run at boundary cell, having a ghost zone can avoid accessing neighborhood chunks. Similar chunking and ghost zone strategy can be applied to gradient computing. Each chunk can also store a small figures used in CNN and other data analysis pipeline, where the size of ghost zone for the chunk can be zero.

Array data access patterns: chunking and ghost zone. Chunking [9] or partitioning an array is a widely used strategy in parallel array processing for aforementioned data analysis operations: convolution and gradient computing. Chunking splits array into contiguous subsets of data and the subsets can be accessed and processed in parallel. An example for a 2D array is presented in Fig. 4, where an $[8X8]$ array is split into four chunks and each chunk has a size of $[4X4]$.

Beyond chunking, another common method in parallel array processing is to access a ghost zone for each chunk. The ghost zone refers to the boundary cells that are built from cells from neighborhood chunks. Having a defined ghost zone avoids communication during the data analysis process. For the above data analysis operations in Equations 1 and 2, each analysis operation on a single cell needs its neighborhood

cells. Hence, when a computation operation is performed on a boundary cell, it may need a cell from other neighborhood chunks. Maintaining a ghost zone for each chunk avoids reading neighborhood cells from other chunks. Both chunking and ghost zone access optimizations using hierarchical storage accelerate array-based data analysis operations.

III. Array Caching in Hierarchical Storage

ARCHIE provides a transparent and efficient caching and prefetching service for array data using hierarchical storage layers. The main goal of this work is to reduce the I/O overhead for data analysis tasks that access array data from both disk-based and SSD-based file systems.

A. Overview of ARCHIE

Towards easing the burden of manually moving data between long-term storage and burst buffer for analysis applications, and using faster SSD-based storage devices efficiently, we propose ARCHIE. HPC applications can transparently use the SSD storage for data analysis applications using ARCHIE.

We show a high-level architecture of the ARCHIE in Fig. 5. ARCHIE is a user-space service that runs in the background and simultaneously with data analysis applications. This overview assumes a disk-based file system and an SSD-based file system. Both of these file systems can be used to store data. ARCHIE provides a cache management function for data analysis applications that read data from disk based file system. In the figure, we highlight the prefetching function of ARCHIE that improves the performance of array-based analysis program by reading ahead data from the disk-based file system to the SSD-based burst buffer. An array is prefetched as data chunks. We show an array with 12 (i.e., $[3X4]$) chunks in the figure. Let's assume that an analysis task is accessing the first two chunks (marked in green) that are read from the disk-based file system. ARCHIE prefetches the following four chunks (shown in blue) into the SSDs for future reads.

Meanwhile, ARCHIE augments each prefetched chunk with a ghost zone layer (shown with red halo around a blue chunk) to match the access pattern on array. A user may specify the width of the ghost zone, which can be zero. The first two chunks (colored white) read into the SSDs are actually empty chunks only containing metadata (e.g., starting and ending offsets). These metadata are written by read function and they provide information for the prefetching algorithm in ARCHIE to predict future chunks. We have implemented parallel prefetching to accelerate parallel I/O of analysis applications. ARCHIE provides fault tolerance mechanism to checkpoint and restart once it failed to handle any potential failures in supercomputer environments.

ARCHIE has the following major components: metadata manager, consistency manager, chunk access predictor, parallel reader/writer, garbage collector, and fault tolerance manager. Functions of these components are described as below:

- **Metadata manager**. ARCHIE manages a metadata table to contain data access information extracted from applications.

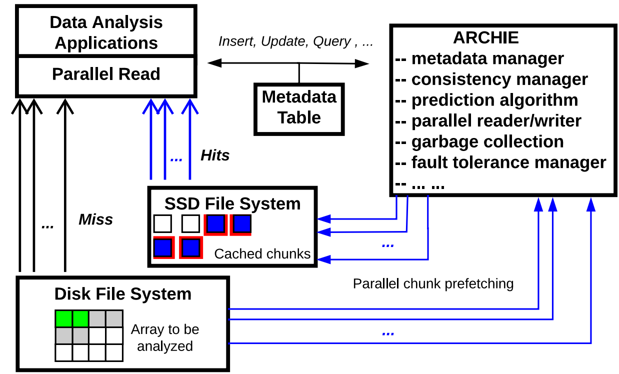


Fig. 5: An overview of ARCHIE in caching data in a hierarchical storage system.

  Entries of the metadata table contain the name of the file being accessed by a data analysis application. This file name information is inserted into the table by the read function linked with analysis applications. We have implemented a HDF5 VOL connector [8] intercept the data read calls from applications. On the other hand, the metadata table is queried by ARCHIE to obtain the file name for prefetch. The metadata table also contains "chunk size", "ghost zone size", and "status of the cached file".
- **Chunk access predictor** uses the information of the cached chunks to predict the future read accesses of analysis applications and to prefetch the predicted data chunks into the faster SSD storage. We provide an expanded description of the prediction algorithm in Section III-D.
- **Parallel reader/writer** brings data from the disk file system into the SSD file system, or vice versa. Because the data to be analyzed may be large, ARCHIE uses parallel reader/writer to prefetch data. Specifically, the reader/writer uses multiple MPI processes that span across all computing nodes that are running the analysis application. These processes concurrently prefetch different chunks from the disks into the SSD to increase the prefetch efficiency.
- **Garbage collector** monitors the usage of the SSD layer. When the capacity of free SSD space is low, the garbage collector service of ARCHIE chooses the oldest chunks and reclaim their space. Algorithms used by garbage collection in ARCHIE is the first-in-first-out (FIFO), which means the oldest chunks are always chosen first for garbage collection. The reason to use the FIFO is that most data analysis programs scan data once and sequentially.
- **Consistency manager**. When a cached chunk is modified by an application, a consistency manager synchronizes updated cached chunks in the storage layers and writes the chunks back to the disk-based file systems. However, since most scientific data is written once and read many times, updates to data rarely occur and hence, the consistency manager function is required infrequently.
- **Fault tolerance manager** records all the actions of ARCHIE and monitors the status of data analysis applica-

tions. When an error occurs in the ARCHIE processes, the fault tolerance manager restarts it and continues from the recorded actions. Meanwhile, if the data analysis programs exit due to errors, the fault tolerance manager cleans up the cache space and the terminates ARCHIE itself.

In the following subsections, we present a detailed functionality of these components, including data chunk movement, using the SSD-based cache, prediction and prefetching algorithms, and fault tolerance.

B. Chunk Management

ARCHIE splits large arrays stored in the disk space into chunks and then prefetches these chunks into the SSD space automatically for data analysis applications. As shown in Fig. 4, the chunks sizes are configured to be aligned with the access granularity of data analysis applications. ARCHIE supports a chunk-based cache management through our new chunk ID calculation method.

Chunk ID calculation. In ARCHIE, an array chunk is a subset of contiguous cells of an array on all dimensions. Theoretically, for a d-dimensional Array with sizes $[D_1, D_2, \ldots, D_d]$, a chunk can be defined as $[l_1 : u_1, l_2 : u_2, \ldots, l_d : u_d]$, where l_i and u_i are lower boundary and upper boundary in i^{th} dimension. Each array chunk is defined using a starting coordinate index and an ending coordinate index in the cache. For example, for a 2D array of 8 rows and 8 columns (i.e., 8 x 8) with chunk sizes of [4 X 4], the first chunk contains cells between coordinate $(0, 0)$ and coordinate $(3, 3)$ (in the row-major format). We use the following hash function to convert the array coordinates into a single integer.

$$CoordHash(i_0, i_k, \ldots, i_d) = \sum_{k=0}^{d-1} \prod_{l=k+1}^{d-1} D_l i_k \tag{3}$$

In Equation 3, D is the size of d-dimensional array and i is the coordinate of a cell in the array. For the above example, the starting coordinates, i.e., $(0, 0)$, are hashed into 0, and the ending coordinates, i.e.,$(3, 3)$, are hashed into 24. The ID of a single chunk in ARCHIE is defined as a string "CoordHash (chunk starting cell coordinate)-CoordHash (chunk ending cell coordinate)". All the cache management functions in ARCHIE are based on the IDs calculated using the hash function shown in Eq. 3. The hash value can also be translated back to array coordinates using the algorithm shown in Fig. 6.

FUNCTION ***CoordHashRevert***$(H, (N_0, \ldots, N_d))$
H: CoordHash value;
$(N_0, \ldots, N_d)$: the size of a d-dimensional array
0. S = H
1. **for** $i \in (d, d-1, \ldots, 2)$ **do**
2. $c_i = H\%N_i$; $H = H/N_i$
3. $c_1 = S$
4. return $(c_1, c_2, \ldots, c_d)$

Fig. 6: Algorithm to convert CoordHash value H back to the d-dimensional array coordinates.

Chunk caching, access controls, and eviction. Since the SSD file systems (such as DataWarp and BurstFS [29]) already provide a global name space integrating all SSD devices in a burst buffer layer, ARCHIE uses binary files created in these file systems to cache the prefetched chunks. Each chunk is stored as a stand-alone binary file in the SSD cache. The names of these binary files are based on the chunk ID, converted as a string. Hence, the file name of a chunk actually contains the starting and the ending cell coordinates of this chunk. We also take advantage of other metadata associated with the file to manage the status of the cached chunks. Each file contains the timestamp to record its creation time, access time, and last modified time. All these timestamps are added by the file system during file creation, access and modification. ARCHIE uses these timestamp to track the life time and last access time of these cached chunks. When the available SSD space is low for fetching a new chunk, ARCHIE chooses the chunk binary file with oldest access time to evict from the SSD space.

ARCHIE also uses the file access permissions (e.g., "S_IREAD" in Linux) to control the status for cached chunks. Once a chunk is prefetched into the SSD space, its corresponding chunk file's access bit is set as *read-only*. When a modification happens to the cached chunk file, the file access permission is switched to be *read-write* by the write function from applications. In implementation, we use the HDF5 VOL connector to intercept the write functions from applications. ARCHIE flushes the dirty chunk files marked as *read-write* to the back-end storage system.

C. Querying Cached Chunks

ARCHIE prefetches data chunks from an array automatically from a disk-based Lustre into a SSD-based DataWarp. For the cached data, ARCHIE manages consistency, as presented in previous subsection. ARCHIE also provides an efficient method for data analysis applications to query the cached chunks in the SSD space. In the current version of ARCHIE, we support three base cases in querying cached chunks. These base queries are presented in Fig. 7. Ideally, a query from application results in an exactly-matched chunk from the SSD space. The first case, "exact match", means that the boundary (i.e., the starting cell coordinate and the ending cell coordinate) of a read area by an application is equal to the boundary of a cached chunk. In the other two base cases, the area of data to be read from applications either belong to a sub-region of a cached chunk or sub-regions of multiple cached chunks. There are also more complex cases where a read request needs multiple full chunks as well as a few partial chunks. These complex cases can be expressed as a combination of the base cases.

D. Parallel Prefetching

ARCHIE is designed to support a diverse set of prefetching algorithms that predict chunks to be read in future by analysis applications. As we mentioned in the previous sections, most data analysis applications scan large arrays sequentially and process the data in chunks. Hence, we have implemented

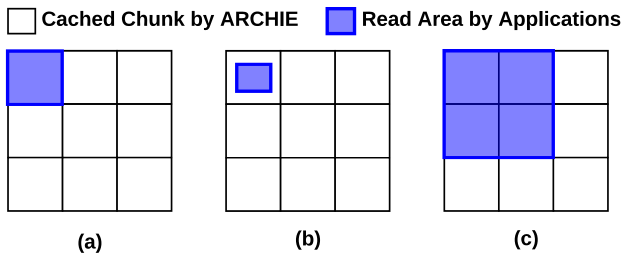


Fig. 7: Three base cases of querying cached array chunks in ARCHIE. a) The read region matches the boundary of a cached chunk. b) The read region is a portion of a single cached chunk. c) The read region is formed by multiple cached chunks. Other cases can be a combination of these base cases in querying cached chunks in ARCHIE.

a prediction algorithm to support sequential array access. ARCHIE converts the ID of a chunk, i.e., "CoordHash (chunk start coordinate)-CoordHash (chunk end coordinate)", into a single integer number. This conversion uses Eq. 3 and Alg. 6. This integer number represents the location of the chunks in an array using row-major order. For example, the first chunk is numbered as 0, the second chunk is numbered as 1, and so on. Since we expect sequential reads from applications, our algorithm identified a pattern of these integer numbers to bring the future chunks into the SSD space. Specifically, we forecast n chunks, where n is the number of processes used for analysis. The number of processes (n) is used because the prefetched chunks need to satisfy all parallel processes for data analysis. These n predicted chunks are prefetched in parallel. The n chunks are prefetched by all ARCHIE parallel processes in a round-robin manner. In the current implementation of ARCHIE, our prefetch function starts to work once it finds the metadata (e.g., file name) from the metadata table. The prefetch function terminates once it finds there are no more chunks to prefetch. Other advanced prefetch algorithms (e.g., regularly striped or randomized) can also be easily integrated into ARCHIE for other access patterns on arrays. We demonstrate the effectiveness of ARCHIE with the current implementation in our evaluation.

E. Ghost Zone Construction

Ghost zone is a small layer attached to the boundary of a data chunk. As stated in the previous section, a ghost zone helps to avoid accessing boundary cells from neighborhood chunks during the execution of an analysis process. Having a ghost zone layer local to processes improves analysis performance significantly. ARCHIE augments each prefetched chunk with a ghost zone. During the prefetch function of ARCHIE, it extends each chunk with ghost zone, which is a novel contribution for caching methods. ARCHIE stores these ghost zones within the same file as the data chunk. When an analysis application wants to read a chunk with ghost zone, it can find the exactly matched chunk in the cache and reads the chunk with ghost zone using a single and contiguous read request.

F. Deadlock Prevention in Collective I/O

Collective I/O [11] is a popular technique used by parallel applications to reduce the number of I/O operations. The semantics of collective operations require all the processes to participate in aggregation. Otherwise, the collective I/O may enter a deadlock state, where one or more processes wait for other processes to join. Hence, data analysis operations may be developed to use collective I/O semantics. In our cache management, the worst case is when some processes can find chunks (i.e., hits) in the SSD cache, and the remaining processes cannot. The processes without any hits will continue to read data from the disks, and the ones with hit do not. If reads are issued with collective semantics, the reads without any hits may wait forever for those with hits. In order to address this issue, ARCHIE introduces a global hit mechanism where either all processes should have hits from the SSD space or all the reads are from disk space.

G. Fault Tolerance

Data analysis application and ARCHIE are two separate programs running concurrently. Either of these two programs could fail due to unforeseen errors introduced by hardware or software, or even themselves. To be tolerant of these errors, ARCHIE monitors the status of the applications during the entire period of runtime. When the data analysis application fails, ARCHIE can automatically terminate itself after the cleanup work that includes writing dirty chunks from the SSD space to the long-term persistent storage, e.g., the disk-based storage. ARCHIE records the progression of its prefetching functions as a journal log, e.g., chunks prefetched and metadata table. If ARCHIE fails, the batch script restarts ARCHIE and it recovers the previously recorded status.

H. Implementation of ARCHIE for HDF5

We have implemented ARCHIE with the HDF5 library [25] and DataElevator [8]. HDF5 is the most popular parallel I/O library that provides an API for users to store and to access array structured data from file system. DataElevator provides a HDF5 VOL connector to accelerate writing data into a hierarchical storage system. Following the same way, ARCHIE extends DataElevator to have an advanced caching and prefetching function for array data. ARCHIE reduces the I/O overhead of data analysis programs, which are read-intensive and use HDF5 I/O. However, ARCHIE can also be implemented within other popular array libraries, such as netCDF and ADIOS [16], to support a wide range of applications. In the current implementation, ARCHIE accepts the user provided hints to specify the chunk size and the ghost zone size. In most cases, we suggest that users can match the chunk and the ghost zone sizes with their data access size in applications. ARCHIE also supports querying chunks with mismatch between access area and cached chunks (as stated above). Users can start ARCHIE as user space service processes (like Data Elevator) distributed on computing nodes for data analysis processes.

IV. Performance Optimizations

In the previous sections, we presented the basic functions of ARCHIE to cache array data in a hierarchical storage system. We now discuss the performance optimizations that ARCHIE brings for data analysis applications.

- **ARCHIE overlaps computation on CPU and data movement across hierarchical storage.** Data analysis applications typically exhibit a pattern of reading and then computing in multiple iterations, where data retrieved and then analyzed. When applications are analyzing the data, the I/O devices tend to be idle. During this idle period, ARCHIE moves the data from the disk space into the faster SSD space. The prefetching operations in ARCHIE are overlapped with the data analysis operations. During the next data read phase, the analysis application can obtain the data from the SSD cache faster than from the disk space. This reduces the end-to-end time for data analysis program by avoiding data retrieval from slow devices. Although, for the very first read phase, the data retrieval happens from the slow devices, the benefit brought by ARCHIE for the latter read phases improves the end-to-end execution performance.
- **ARCHIE converts non-contiguous reads from storage devices into contiguous ones.** As we stated above, ARCHIE uses individual files for storing each cached chunk prefetched from a large array. Hence, data analysis applications can obtain a cached chunk by reading a single and contiguous binary file if the boundaries of a request match with the cached chunk. Since array data are mostly linearized on storage device, reading chunks from the original multi-dimensional array tends to have large number of small and non-contiguous reads [9]. These reads may span across different rows and results in large overhead. ARCHIE can avoid this type of I/O overhead by redirecting application to read contiguous binary files. Thus, based on the overlap method in previous paragraph, ARCHIE can further reduce the I/O cost of array data based analysis applications.

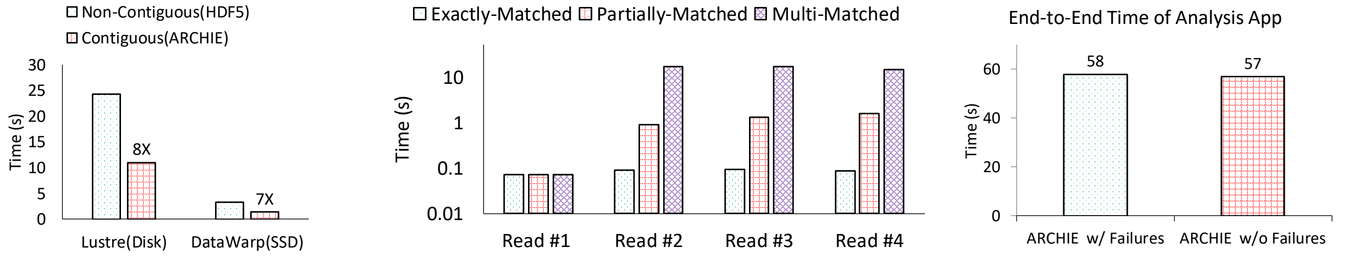
V. Experimental Results

We evaluated performance of ARCHIE on a Cray XC40 supercomputer, named Cori, installed at NERSC. Cori has 2,388 nodes with Intel Xeon (Haswell) processors and 9,688 nodes with Intel Xeon Phi (Knight's Landing - KNL) processors. Cori has a 1.8 PB SSD-based burst buffer managed by Cray DataWarp with a peak performance of 1.7TB/sec. Its disk-based storage system is managed by Lustre, which has 28 PB space and can deliver up to 700 GB/sec performance. We compare ARCHIE with Lustre to show the benefits of the prefetching function on accelerating I/O operations by converting non-contiguous I/O to contiguous ones. DataWarp provides tools, e.g., "stage-in", to manually move a data file from disk space into SSD space. In contrast, ARCHIE automatically prefetches the array data from Lustre to the burst buffer in chunks. Our method avoids application's wait time for the entire file data movement at the start of job execution enforced by DataWarp. Our tests demonstrate both detailed performance metrics (i.e., time) for a single read as well as the end-to-end time. The end-to-end time is the total I/O time that data analysis programs spent in reading data.

A. Evaluation with micro-benchmarks

This section reports our evaluation of reading data from a 2D array data. The size of the 2D array is 256K x 256K, where K = 1024 and it contains floating point values. The data analysis operation is a 2 x 2 convolution operation. By default, we used 256 processes (i.e., one process per CPU core) to perform the data analysis. The 2D array is split into chunks with a size of [8192, 8192]. The size of the ghost zone is [1,1] to keep convolution operation locally at each process. We use 256 processes to do the cache management and prefetching work. Based on this configuration, the array has 1024 chunks in total and these chunks can be read by 256 analysis processes in four batches. We denote the batches as "Read#1", "Read#2", "Read#3" and "Read#4".

- **Converting non-contiguous accesses into contiguous accesses on both disks and SSDs.** In Fig. 8a, we show the performance improvement with ARCHIE's converting non-contiguous accesses to contiguous ones. As stated above, this test uses an analysis framework, called ArrayUDF [9], to perform convolution computing on a large 2D array. ArrayUDF accesses the array in chunks. ARCHIE prefetches the chunks from original multi-dimensional array and then stores each chunk as an individual binary file. In the original array structure in HDF5, rows of a multi-dimensional chunk tend to be scattered in different locations of storage devices. In contrast, ARCHIE stores all these scattered rows into a single binary file. Data analysis can read the whole chunk as a contiguous one increasing locality. This improved locality achieves $\approx 8X$ performance over Lustre and $\approx 7X$ over DataWarp. These test results show that ARCHIE can improve the I/O performance of data analysis applications on both disk- and SSD-based file systems.
- **Overhead of querying chunks.** ARCHIE allows applications to query cached chunks in the SSD space. We evaluated the overhead of the three base query cases mentioned earlier and report the execution time in Fig. 8b. When a query accesses the exactly matched chunks, the overhead for ARCHIE is small, i.e., less than 0.1 seconds. For the partially matched cache chunks, the overhead of ARCHIE is higher. Because partially matched chunks in ARCHIE need to go through all the cached chunks once but the exactly matched cache only checks the desired data chunks. For the "multi-matched"-case, where a few cached chunks are needed, ARCHIE needs to go though all the cached chunks multiple times. Therefore, the overhead increases again. For the first read of application running, it is usually a miss in ARCHIE and therefore it does not involve any metadata checking overhead.
- **Fault tolerance.** To test the fault tolerance of ARCHIE, we manually injected a forced termination to ARCHIE by adding an `exit(-1)` function in an arbitrary location. In Fig. 8c, we show the performance of running the data



(a) Non-contiguous vs. contiguous read.

(b) Overhead of querying cached chunks.

(c) Restart ARCHIE to resist error.

Fig. 8: Performance of ARCHIE with micro-benchmarks.

analysis program using ARCHIE with a forced failure and without one. When ARCHIE terminates during runtime, it can restart itself and continue previous operations recorded in the journal log. Test results show that ARCHIE can successfully restart itself and the failure and restart negligible impact on the end-to-end time of the data analysis program.

- **Comparing with Lustre and DataWarp.** In Figure 9, we compare the I/O performance of using Lustre and DataWarp with that of ARCHIE to serve the read requests of the convolution analysis program. For the test with Lustre, the analysis program reads all data from the disk space. As expected, it has the worst performance due to the slow disk performance. DataWarp requires moving the entire data files being read into the SSD-based burst buffer and then the application's reading from the SSDs. However, the application's reads can only start after all the files are moved to SSDs. The read time from SSDs, which are faster than disks, is smaller than that from Lustre. In total, i.e., bringing data from Lustre and then reading from the burst buffer, the DataWarp case is $1.5\times$ faster than Lustre For ARCHIE, the first read is slightly costlier than reading data directly from Lustre, as ARCHIE has to observe data accesses and inform the application to read data directly from the disks. The following reads are much faster than both the Lustre and Datawarp cases because the application can read the prefetched data by ARCHIE. The I/O time is further reduced by converting non-contiguous I/O requests into contiguous ones. We have tested using the disks as a cache for ARCHIE, and the overall improvement of contiguous accesses for the four read requests is $2.6\times$ over accessing data directly from Lustre. Using the SSD burst buffer, ARCHIE achieves $3.3\times$ overall performance benefit.

B. Evaluation with Real-world Scientific Applications

We describe performance evaluation of using ARCHIE in supporting three real-world data analysis tasks on a hierarchical storage system. These data analysis tasks are implemented in C++ and ARCHIE provides a transparent method through HDF5 to accelerate their I/O operations. We use Lustre and DataWarp for comparing I/O performance.

Convolution analysis of a climate dataset. As mentioned in Section II-B, convolutional neural network (CNN)-based analysis on CAM5 data is used to detect extreme weather events [21]. In our evaluation of ARCHIE, we focus on the most time-consuming step, i.e., convolution computing, in CNNs. The CAM5 dataset used in this test is a 3D array with size $[31, 768, 1152]$, where 768 and 1182 are the latitude and the longitude dimensions, respectively, and 31 is the number of height levels from the earth into the atmosphere. The filter size for the convolution is $[4, 4]$. The chunk size is $[31, 768, 32]$, resulting in a total of 36 chunks. The analysis application runs with 9 MPI processes. We run ARCHIE runs with another 9 processes. In this configuration, there are four batches of reads and each batch accesses 9 chunks. We present the read time of the analysis application in Fig. 10. Reading all the data from the disks, i.e., Lustre, gives the worst performance, as expected. Using DataWarp to move the data from Lustre to the burst buffer reduces the time by 1.7X for reading data but it contains initial overhead in staging the data in the burst buffer. Using the ARCHIE cache on both the disks and on the SSDs

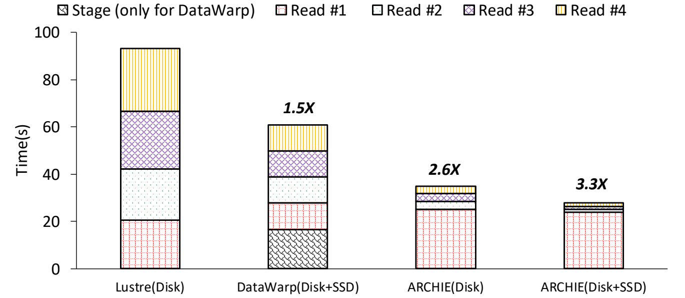


Fig. 9: Comparison of Lustre, DataWarp, and ARCHIE in serving data read for the convolution analysis program.

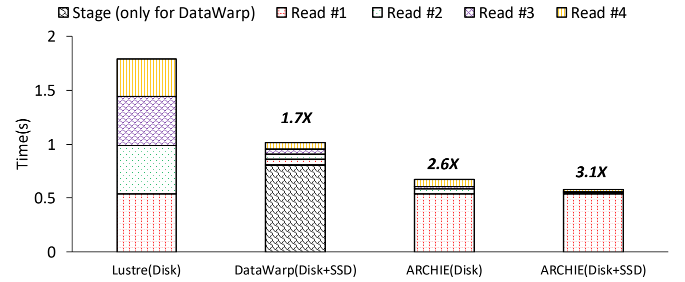


Fig. 10: Evaluation of ARCHIE with CAM5 dataset

reduces the time for reading data. The advantage of ARCHIE comes from converting non-contiguous reads into contiguous reads. ARCHIE can also prefetch data to be accessed in future into the burst buffer as chunks and achieves the best performance. Overall, for the CNN use case, ARCHIE is $3.1\times$ faster than Lustre and $1.8\times$ faster than DataWarp.

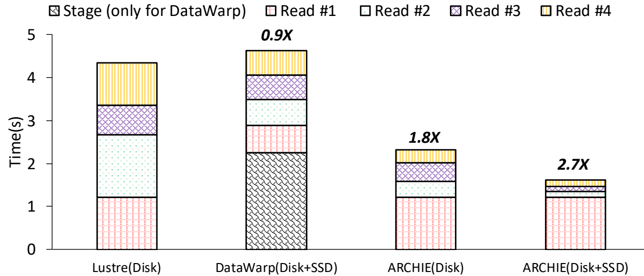


Fig. 11: Evaluation of ARCHIE with plasma physics data

Gradient computation of plasma physics dataset. In this test, we used a 3D magnetic field data generated by a VPIC simulation [3]. The data is a 3D array with size as $[2000, 2000, 800]$. The data analysis operation is gradient computing, which is defined in Eq. 2. The chunk size for parallel array processing is $[250, 250, 100]$, giving a total of 512 chunks. Our tests use 128 processes to run the analysis program and use only 64 processes to run ARCHIE. In this test, we have a ghost zone with a size of $[1, 1, 1]$. We present the read time of this analysis in Fig. 11. The performance comparison have the same pattern as the that of convolution on CAM5 data. Overall, ARCHIE is $2.7\times$ faster than Lustre and $2.4\times$ faster than DataWarp.

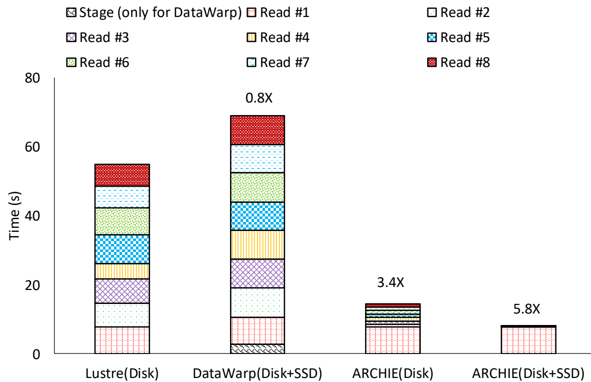


Fig. 12: Evaluation of ARCHIE with combustion (S3D) data

Vorticity computation on combustion data. S3D simulation code captures key turbulence-chemistry interactions in a combustion engine [5]. An attribute to study the turbulent motion is vorticity, which defines the local spinning motion for specific location. Given the z component of the vorticity at a point (x, y), the vorticity analysis access four neighbors for each point at $(x+1, y)$, $(x-1, y)$, $(x, y-1)$ and $(x, y+1)$, as defined in [9]. Our tests use a 3D array with size $[1100, 1080, 1408]$. The chunk size is $[110, 108, 176]$ and the ghost zone size is $[1, 1, 1]$, giving 800 chunks. We use 100 processes to run analysis programs and 50 processes to run ARCHIE. The read performance comparison is shown in Fig. 12. The analysis program reads all 800 chunks in 8 batches. ARCHIE outperforms Lustre by $5.8\times$ and is $7\times$ better than DataWarp. In this case, DataWarp performs $1.2\times$ slower than Lustre.

In summary, our performance evaluation of three scientific analysis kernels shows that ARCHIE accelerates data analysis through its cache management and the prefetching function by taking advantage of hierarchical storage system. The performance benefit of ARCHIE primarily depends on the data size, the number of non-contiguous accesses, and the number of iterations data is accessed.

VI. Related Work

Existing work on cache management can be classified into three categories: single-node cache management, distributed cache management, and cache management on the hierarchical storage. Early efforts, such as LRU, LFU, FIFO, CLOCK [24], were primarily dedicated to single-node cache management. These algorithms, as well as their ramifications [18], [20] have been used for cache management on various devices, such as disk cache on DRAM and on SSDs. These algorithms also serve as the building blocks for distributed and hierarchical cache management.

Numerous distributed caching systems have been developed to tackle the cloud computing I/O workloads, such as Memcached [19]. These caching systems typically expose to their upstream applications key-value based interfaces, and allow users to quickly retrieve the data of interests based on their keys. Different from these work, ARCHIE is designed to handle the HPC I/O workloads, in which data are typically formatted as multi-dimensional arrays.

Recently, SSD-based burst buffers have been widely deployed on supercomputers to provide temporary caching service for HPC I/O workloads. A large body of burst buffer management software is designed to move data only between burst buffer and applications [15], [31], [26], [28], [30]. On the other hand, burst buffer software such as DataWarp [6], DDN IME [7], IBM Spectrum scale [12], and Data Elevator [8] support transparent caching feature that redirects writes to the PFS to burst buffer transparently, and asynchronously flushes data to the PFS. However, this feature is mainly designed for accelerating bursty writes. In order to provide read caching, users have to manually populate data into burst buffer. All of these systems cache data in the form of flat binary files, without considering their multi-dimensional data structure. Recent works such as Hermes [13] and UniviStor [27] provides multi-tiered I/O buffering system for flat file. Different from these work, ARCHIE provides a multidimensional array semantic-aware caching and prefetching that allows users to utilize the hierarchical storage system.

A few recently proposed cache management systems have the prefetching feature. Eley [33] and [34] prefetches the input data of applications into SSD to reduce the latency of reading

data inputs. Byna [2] and Tang [24], [23] propose prefetching algorithms that fetch data based on the historical access patterns of flat files. In contrast, ARCHIE works efficiently on multi-dimensional arrays to accelerate read operations of analysis applications using hierarchical storage systems.

VII. Conclusions

Scientific data analysis applications often operate on massive amounts of data. Retrieving data from disk-based file systems often performs poorly due to the slow mechanical components of the hard disk drives. Augmenting the disk-based storage layer with a multiple layers of non-volatile storage has been considered a cost-effective solution to this problem. However, software solutions to integrate the hierarchical storage system for efficient data retrieval is either unavailable, or needs significant manual involvement. To address this issue, we have proposed ARCHIE to provide an efficient caching and prefetching functionality for hierarchical storage systems. We introduce optimizations such as prefetching data from multi-dimensional arrays that are common data structures in scientific data. Our performance evaluation with micro-benchmarks and with real scientific data analysis operations demonstrate that ARCHIE improves I/O performance significantly. In the case of vorticity computation of combustion data, ARCHIE outperforms Lustre and DataWarp by more than $\approx 5\times$. We plan to design more generic and advanced prediction algorithms for ARCHIE to improve prefetching accuracy further.

Acknowledgment

This effort was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231 (program manager Dr. Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science user facility.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, et al. TensorFlow: A System for Large-scale Machine Learning. In *OSDI*, 2016.

[2] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC 2008*.

[3] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu. Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation. In *SC '12*.

[4] M. Chaarawi and Q. Koziol. HDF5 Virtual Object Layer. Technical report, Available: https://confluence.hdfgroup.uiuc.edu/display-/VOL/Virtual+Object+Layer, 2011.

[5] J. H. Chen, A. Choudhary, B. de Supinski, and et al. Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D. *Computational Science & Discovery*, 2(1), 2009.

[6] Cray. DataWarp User Guide S-2558-5204. Technical report, Available: http://docs.cray.com/books/S-2558-5204/S-2558-5204.pdf, 2016.

[7] DDN. DDN IME Web Page. http://www.ddn.com/products, 2014.

[8] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, and N. Keen. Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. In *HiPC*, 2016.

[9] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu. ArrayUDF: User-Defined Scientific Data Analysis on Arrays. In *HPDC*, 2017.

[10] D. Fellinger. The State of the Lustre File System and The Lustre Development Ecosystem. Technical report, 2003.

[11] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre File Systems. In *IASDS10*, 2010.

[12] IBM. IBM Spectrum Scale. https://www.ibm.com/us-en/marketplace/scale-out-file-and-object-storage, 2018.

[13] A. Kougkas, H. Devarajan, and X.-H. Sun. Hermes: A Heterogeneous-aware Multi-tiered Distributed I/O Buffering System. In *HPDC*, 2018.

[14] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD*, 2017.

[15] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *MSST*, 2012.

[16] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, et al. Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Frameworks. *Concurr. Comput. : Pract. Exper.*, 26(7):1453–1473, May 2014.

[17] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *IPDPS*, pages 1–10. IEEE, 2009.

[18] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*, 2003.

[19] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *NSDI*, 2013.

[20] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[21] E. Racah, C. Beckham, T. Maharaj, Prabhat, and C. J. Pal. Semi-Supervised Detection of Extreme Weather Events in Large Climate Datasets. *CoRR*, 2016.

[22] F. Schmuck and R. Haskin. GPFS: A Shared-disk File System for Large Computing Clusters. In *FAST'02*, 2002.

[23] H. Tang, S. Byna, S. Harenberg, W. Zhang, X. Zou, et al. In situ storage layout optimization for amr spatio-temporal read accesses. In *45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016.

[24] H. Tang, X. Zou, J. Jenkins, D. A. Boyuka, S. Ranshous, D. Kimpe, S. Klasky, and N. F. Samatova. Improving read performance with online access pattern analysis and prefetching. In *Euro-Par*, 2014.

[25] The HDF Group. HDF5 user guide. http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.user.html, 2010.

[26] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin. Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines. In *FAST'13*, 2013.

[27] T. Wang, S. Byna, B. Dong, and H. Tang. UniviStor: Integrated Hierarchical and Distributed Storage for HPC. In *CLUSTER*, 2015.

[28] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. An Ephemeral Burst-buffer File System for Scientific Applications. In *SC*, 2016.

[29] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. An Ephemeral Burst-Buffer File System for Scientific Applications. In *SC*, pages 807–818. IEEE, 2016.

[30] T. Wang, A. Moody, Y. Zhu, K. Mohror, K. Sato, T. Islam, and W. Yu. MetaKV: A Key-Value Store for Metadata Management of Distributed Burst Buffers. In *IPDPS*, pages 1174–1183. IEEE, 2017.

[31] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. BurstMem: A High-Performance Burst Buffer System for Scientific Applications. In *Big Data*, 2014.

[32] M. Wehner, Prabhat, K. A. Reed, D. Stone, W. D. Collins, and J. Bacmeister. Resolution Dependence of Future Tropical Cyclone Projections of CAM5.1 in the U.S. CLIVAR Hurricane Working Group Idealized Configurations. *Journal of Climate*, 28(10):3905–3925, 2015.

[33] O. Yildiz, A. C. Zhou, and S. Ibrahim. Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC Systems. In *CLUSTER*, 2017.

[34] W. Zhang, H. Tang, X. Zou, S. Harenberg, Q. Liu, S. Klasky, and N. F. Samatova. Exploring memory hierarchy to improve scientific data read performance. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 66–69, 2015.