UniviStor: Integrated Hierarchical and Distributed Storage for HPC

Teng Wang, Suren Byna, Bin Dong, Houjun Tang
Lawrence Berkeley National Laboratory
Email: {tengwang, sbyna, dbin, htang4}@lbl.gov

***Abstract*—**
High performance computing (HPC) architectures have been adding new layers of storage, such as burst buffers, to tolerate latency between memory and disk-based file systems. However, existing file system and burst buffer management software typically manage each storage layer separately. As a result, the burden of moving data across multiple layers falls upon HPC system users. To hide the complexity of managing the scattered storage devices from applications, we introduce *UniviStor*, a data management service offering a unified view of storage layers. By considering each layer's distinct characteristics, UniviStor provides performance optimizations and data structures tailored for distributed and hierarchical data placement, interference-aware data movement scheduling, adaptive data striping, and lightweight workflow management. UniviStor supports parallel I/O library APIs, such as MPI-IO and HDF5. Our evaluations on a large-scale supercomputer demonstrated that UniviStor outperforms Data Elevator, a state-of-the-art transparent caching solution for burst buffers by up to 17×, and Lustre by up to 46×.

I. Introduction

Upcoming exascale storage sub-systems in HPC are adding new layers of heterogeneous storage devices. To improve the I/O performance on HPC systems that are traditionally limited by slow disk-based devices, the storage subsystem is being expanded in both hierarchical and distributed manner. Hardware such as NVRAM-based burst buffer is added on each compute node, or on I/O nodes accessible by all compute nodes, or on both. In addition, faster DRAM and Storage-Class Memory (SCM) are being added on compute nodes.

Efficient use of the deep memory and storage hierarchy is a complex task for application developers and HPC system users without a unifying storage sub-system view. Existing storage management solutions are generally designed to manage one or a few layers, and require applications or users to explicitly move data across layers using each layer's individual solutions. For instance, data in node-local storage such as DRAM and SSDs are addressable within the compute node containing them using its local operating system. To share data across nodes, libraries such as UPC [1], OpenShmem [2], and DataSpaces [3] provide a global address space for moving data across node-local DRAM. They expose to applications with library-specific interfaces. On the other hand, shared burst buffer has a larger capacity than node-local storage and is connected to all the compute nodes via a high-speed network; its data management software, such as Cray DataWarp [4] and DDN IME [5] expose to applications POSIX and parallel I/O interfaces (e.g. MPI-IO [6], HDF5 [7], and netCDF [7], [8]). The distinct interfaces and data services for each layer make data management a complex task. For instance, a memory-hungry application using HDF5 may want to place part of its file on DRAM and spill the rest to the shared burst buffer. Using existing solutions, the application developer has to resolve the interface incompatibility and to explicitly invoke the service for different layers. DAOS [9] is an object-based storage system being developed as a replacement for Lustre to unify different storage devices, but its current deployment requires significant storage infrastructure changes. Hermes [10] is another recent I/O buffering system developed to automate data movement across storage layers, but current design relies on the fact that users know the behavior of their applications in advance.

The data management task is further complicated by the transiency of data on node-local storage and shared burst buffer, as they are typically allocated to a job on-demand and data integrity is assured within the job's life cycle [11]. Consequently, important data have to be flushed to a parallel file system (PFS) for long-term persistence. Transparent caching [12] has been either proposed or supported by several burst buffer software [4], [13], [14] to enable an I/O redirection between burst buffers and PFS. Likewise, these solutions are only designed for flushing data from one layer to PFS without an integrated solution considering all available layers.

Providing a unified storage view integrating memory and storage layers is challenging. First, a unified address space is needed for data in all layers. Second, each layer has distinct performance characteristics, an integrated system has to be optimized for each layer. For instance, DRAM or SCM has high performance that is sensitive to context switches and task placement among cores. In contrast, shared storage layers, such as shared burst buffer and PFSs are vulnerable to the I/O contention [15], [16], [17] caused by multiple processes concurrently accessing the same file.

To ease the burden of rewriting existing codes, an integrated storage solution should be compatible with standard parallel I/O libraries, such as MPI-IO, HDF5 and netCDF. While existing software generally supports standard I/O on the shared burst buffer, there is still a lack of counterpart support on the node-local DRAM/burst buffers. Furthermore, a core benefit of the DRAM and burst buffer layer is the acceleration of scientific workflows with in-situ/in-transit analysis, where analysis programs can immediately read data that are close

2168-9253/18/$31.00 ©2018 IEEE
DOI 10.1109/CLUSTER.2018.00025

to computing nodes. However, the aforementioned parallel I/O libraries do not support this feature, forcing analysis programs to read only after a simulation completes or requiring an application-specific workflow management implementation. Although ADIOS [18], [19] supports in-situ/in-transit analysis using DataSpaces/FlexPath [20], applications based on other I/O libraries have to be rewritten with ADIOS' I/O interface.

Towards providing a unified view of various storage layers, we have designed and implemented *UniviStor*, a system that exposes the distributed and hierarchical storage spaces to applications as a single mount point. UniviStor adopts the design philosophy of several state-of-the-art data service systems [21], [22], [23], [24] that decouple address management from data management, and implement the address management as a distributed key-value service. More importantly, it extends this philosophy with a unified address management for different layers and location-aware data service. Furthermore, we have developed an interference-aware resource scheduling procedure that accelerates writing and reading data using the DRAM/SSDs distributed on compute nodes. We have introduced adaptive data striping for load balanced data movement to the disk-based PFS. UniviStor provides this unified service via the MPI-IO interface, which is used by high-level I/O libraries such as HDF5. We have also added parallel I/O support for in-situ/in-transit analysis on DRAM/burst buffers with a lightweight workflow management. The research contributions of this paper include:

- Design and implementation of UniviStor to integrate node-local and shared storage devices into a unified storage space using a distributed metadata service to manage the address space. UniviStor provides compatibility with existing I/O APIs and hides the complexity of managing different storage layers.
- Performance optimization strategies to support distributed and hierarchical placement of data using log-structured writes and interference-aware data movement scheduling for fast node-local caching.
- An analytical performance model to guide adaptive data striping on a parallel file system.
- A lightweight mechanism to orchestrate applications with data access dependencies when data are distributed across a multi-layer storage subsystem.

We have evaluated UniviStor using both benchmarks and application I/O workloads, and compared our system with the state-of-the-art solutions, including Data Elevator and Lustre, on a production HPC system. Our experiments demonstrate that UniviStor integrates both node-local and hierarchical storage layers efficiently. UniviStor outperforms Data Elevator and Lustre by up to $\approx 17\times$ and $\approx 46\times$, respectively. We have also evaluated the performance of UniviStor in using various combinations of storage hierarchy to support data generation and analysis workflow.

In the remainder of the paper, we introduce UniviStor architecture and optimization strategies in Section II. In Section III, we first evaluate UniviStor with optimizations using microbenchmarks, and then compare its performance with Data Elevator and Lustre file systems using both microbenchmarks, I/O kernels from scientific simulations, and workflows. We then discuss related work (§IV) and conclude the paper (§V).

II. UniviStor Integrated Storage

A. Overview

We show the high-level architecture of our proposed integrated storage system in Fig. 1. The memory and storage layers shown in the figure include local DRAM and/or NVRAM-based burst buffer on each compute node (CN), shared SSD-based burst buffer (on specialized nodes accessible by all compute nodes), as well as a disk-based Parallel File System (PFS). UniviStor services include managing metadata, caching, and moving data in the hierarchy.

UniviStor server processes are launched as a parallel program on all the compute nodes allocated to an application job. This job can include one or more coupled parallel client applications (e.g., App 1 and 2 in Fig. 1). The number of servers on each node is configurable by users and is set to a default value of 1. The UniviStor servers collectively serve the I/O requests from the client applications, and manage the hierarchical storage space with several services. **Data caching service** temporarily cache clients' writes to the fastest available memory or storage device, including node-local DRAM or SSDs, shared burst buffer. If the dataset is too large to fit in DRAM and burst buffer, data is written to disk-based PFS.

When a client process requests a portion of data, each client process's read requests are directed to the server process co-located on the same compute node. The server reads the requested data from the storage space and returns the data to the client. Delegating the read requests to the co-located server process enables data sharing among processes in different programs. In Fig. 1, processes in App 1 and App 2 can share data with the help from the server processes running on all compute nodes. In order to find the requested data in the hierarchical storage space, a distributed **metadata service** is provided by all the servers for data look-up. Server-side **flush service** is triggered by the client program automatically at the file close time, in which servers collectively flush the cached data to a PFS for long-term data persistence. An application without data persistence requirement can optionally disable the flush service.

UniviStor clients are parallel applications launched in the same job as the UniviStor servers (i.e., App 1 and App 2 in this example). Each application is linked with the UniviStor library, which transparently redirects their parallel I/O requests to UniviStor servers. These requests can be issued by MPI-IO, or high-level I/O libraries such as HDF5. I/O redirection has two benefits. First, redirecting read/write requests to UniviStor servers' faster storage allows different applications to share data directly on the high throughput storage layers, avoiding the interaction with the slower disk-based storage. Second, redirecting the file close requests to UniviStor servers triggers automatic data flush operations, during which UniviStor servers asynchronously move the cached data to the long-term

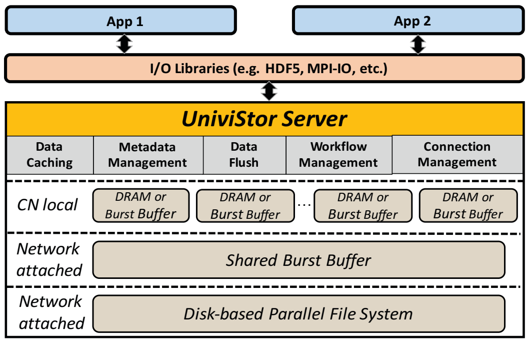


Fig. 1: A high-level view of UniviStor'storage hierarchy.

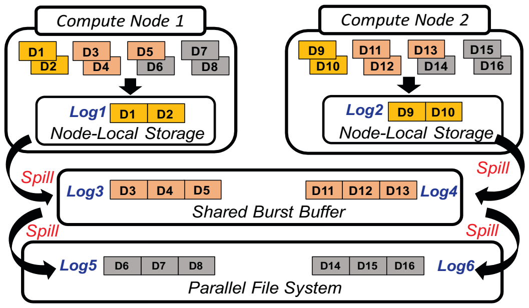


Fig. 2: Distributed and hierarchical data placement.

storage layers concurrently while the application processes continue to perform their computation. However, redirecting all the I/O requests without considering their data dependency can incur consistency issues. To avoid this issue, we implemented UniviStor with a lightweight workflow management component for coordinating the order of data accesses from different client applications.

We have implemented UniviStor client as an I/O driver in the MPI-IO layer. Users can enable this I/O driver by setting the environment flag "ROMIO_FSTYPE_FORCE" as "UniviStor". In order to use UniviStor, users start the UniviStor server program before launching the client applications. UniviStor servers automatically detect applications' membership with its connection management module that handles clients' connection and disconnection requests sent from `MPI_Init` and `MPI_Finalize`, respectively. The server processes terminate after all the client applications exit. In the following subsections, we describe UniviStor's data management strategies (§II-B), interference-aware scheduling of the servers (§II-C), adaptive data striping (§II-D), and lightweight workflow management (§II-E).

B. Data management in the integrated storage

UniviStor provides an integrated view of both distributed and hierarchical storage spaces. We make four design choices to achieve this goal: distributed and hierarchical data placement, virtual addressing, distributed metadata service, and location-aware read service.

1) Distributed and hierarchical data placement: We have designed a Distributed and Hierarchical data Placement (DHP) strategy to efficiently place data on the multi-layer storage space. Using DHP, when a client process opens a file for writing, a memory-mapped log-structured file is created (via `mmap`) in DRAM. The size of the file is configurable by applications. The file persists beyond the client process's life cycle in the form of shared memory managed by UniviStor. Data is first written to this file until its allocated space depletes. When space is not available in the memory-mapped file, a new log file is created in the next available storage layer (e.g., node-local storage or shared burst buffer) and the subsequent data is written to it until exceeding the new file's allocated capacity. This operation repeats across all the storage layers until data reaches the destination storage layer set by the application, which is typically a disk-based PFS.

We illustrate DHP in Fig. 2 with an example. Let D1-D16 be the data segments laid out sequentially within a single logical shared file, issued by two client MPI processes located on two compute nodes. Assuming each segment is sized 1, the capacity of each log on the node-local storage and a shared burst buffer are 2 and 3, respectively. With DHP, segments from each process are physically stored in different logs spread across three storage layers. This design has two benefits: first, it transforms the "shared write" pattern into "file-per-process" write pattern. In Fig. 2, D1-D16 originally belong to the same shared file, but are physically stored as log files for each process. This transformation accelerates writing to the shared burst buffer and the PFS with reduced I/O contention [25], [26]. Second, this approach takes advantage of the storage space available on all storage layers.

By default, UniviStor configures the capacity of each log as c/p, where c is the capacity and p is the number of processes. For node-local storage, c is the available capacity of the log's local storage, p is the number of client processes launched on its local node. In the shared storage layer (e.g. shared burst buffer), all devices are shared by all the clients across nodes. Hence, c is the available capacity of all storage devices in this layer, p is the total number of client processes in the parallel application(s).

Internally, the storage space of each log file is formatted as a set of data chunks. Data are appended inside each chunk in a log-structured manner. This log-structured writes can maximize the bandwidth of both disk-based PFS and SSD-based burst buffer with the sequential write pattern. UniviStor also creates a *free chunk stack* for each log file. This stack records all the free chunk IDs. Once a chunk is used up, a new chunk ID is popped up from the stack and data are written to the corresponding chunk. Once a chunk is overwritten/deleted, its ID is pushed back to the stack for reuse.

2) Virtual addressing: As DHP places data in different layers of storage, reading data remains a challenge because segments issued by each process may reside on logs belonging to different storage layers. Locating the storage device of a requested segment requires a global addressing scheme that spans across multiple layers. In order to uniquely locate a segment among the individual process's log files, we introduce the concept of a *Virtual Address* (VA). The VA of a segment

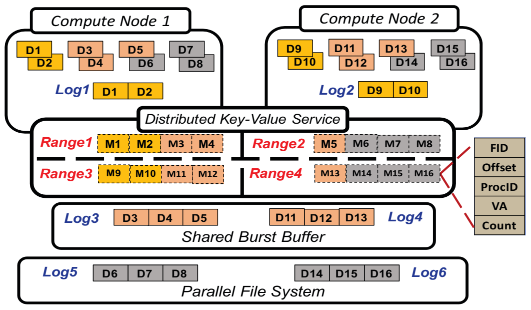


Fig. 3: Distributed metadata service.

located on ith storage layer is defined in Eq. 1.

$$VA_i = \sum_{k=0}^{i} C_i + A_i \tag{1}$$

In Eq. 1, C_i represents the capacity of a log file in the i^{th} storage layer. A_i denotes the physical address of a segment within that log file. Using the example in Fig. 2, segment D4's physical address in Log3 is 1, its VA is 3. VA identifies both the storage layer of a segment and its physical address within its log file on that layer.

3) Distributed metadata service: The adoption of VA itself is insufficient for reading data. On the one hand, the VA of each segment is only available for the logs of the producer (or source) process and segments generated by different processes can have the same VA. For instance, in Fig. 2, D4 and D12 are produced by source processes on Nodes 1 and 2, respectively. They are indistinguishable since both have a VA of 3. On the other hand, using log-structured write, VA of a segment is not the same as its offset in a logical file. For instance, in Fig. 2, D12's logical offset is 12 in the logical shared file, but its VA is 3. Due to this difference between logical offset and VA, UniviStor requires to maintain a structure that maps each segment's offset to its VA and the source process information.

A naïve solution is to record the mapping information for each file segment on a global map, and store the map on one of the server processes. However, such centralized approach is not scalable as the server containing the map becomes a bottleneck. Instead, UniviStor stores this map using a distributed key-value (KV) store maintained by all UniviStor servers. In Fig. 3, a metadata record is created for each file segment (M1-M16 for D1-D16). Each record contains several attributes that associate its segment's logical offset with its VA and source process. In Fig. 3, FID and offset locate the logical file of a segment and its offset in the file, respectively. ProcID and VA point to the source process and its virtual address, respectively. These records are distributed on different servers based on their logical offsets. For instance, in Fig. 3, M1-M16 are partitioned into 4 ranges based on their offsets (1-16), and these ranges are assigned to servers on two compute nodes in a round-robin manner. Consequently, M1-M16 are distributed to the servers based on their belonging ranges.

With this distributed metadata service, when a server receives a read request from its co-located client process (e.g. request for D12 on Node 2 in Fig. 3), it looks up this segment's source process (process on Node 2) and VA (3 for D12), and forwards the request to a remote server (e.g., a server on Node 2). Upon receiving the read request, the server on Node 2 retrieves the requested file segment based on its VA, and returns it to the requesting server, and the requesting server then delivers the segment to its client.

4) Location-aware read service: The read service assumes that each log file is only visible within its host compute node. For instance, in Fig. 3, D10 resides on the local storage of Node 2, its log file (Log2) is only visible to Node 2. In this case, each read request is directed to the server whose node contains the requested segment (e.g. Node 2 for D10). Consequently, each read request incurs at least one round-trip of data transfer across the network. However, the requested segment can also reside on the shared burst buffer (e.g. D12 in Fig. 3), with its host log globally visible to all compute nodes, the requesting client can directly retrieve the segment from the shared burst buffer, the aforementioned read service incurs additional data transfer cost. Furthermore, this read service can also introduce additional memory copy overhead if the requested segment resides on the local storage, since the client's read request goes through the co-located server process.

To further improve read performance, we have designed a location-aware read service. To avoid the memory copy overhead, each UniviStor server maintains a shared metadata buffer in its local storage that caches the metadata records for all the locally generated segments. When the client issues a read request, it first compares the requested segment with the locally cached metadata. All the portions of requested segment cached locally are directly retrieved from the local storage, without going through the co-located UniviStor servers. To avoid additional data transfers among UniviStor servers, we allow the client processes to directly retrieve the metadata of the requested file segment. Once the client acquires the metadata, it can distinguish the portion of the segment that resides on the shared burst buffer, and directly retrieves this portion from the shared burst buffer without transferring data between the remote servers.

C. Interference-aware resource scheduling for local caching

UniviStor accelerates applications by directing their I/O requests to the node-local memory or storage space. Performance benefit of UniviStor depends on how application processes can exploit the multi-core parallelism for parallel memory accesses. Existing HPC systems typically adopt Linux's default task scheduler (i.e. Complete Fair Scheduler (CFS) [27]) to schedule processes among the cores. Although CFS works well for the web servers and desktop applications that have the intermittent and random arrival of I/O requests, it is not optimized for the highly concurrent and synchronized scientific workloads. In Fig. 4(a), we show the potential issues with CFS, where 6 processes are launched on the same compute node with two NUMA sockets and 6 cores (C1-C6). P1_1 and P1_2 are client processes from Application 1. P2_1 and P2_2 are client processes from Application 2. These four processes

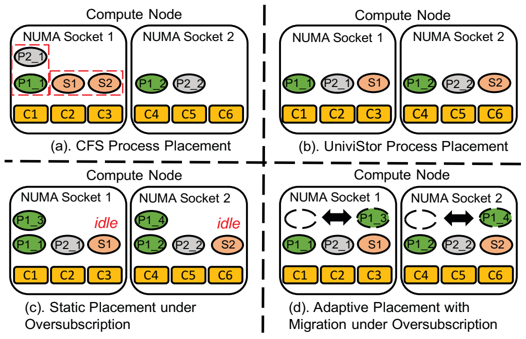

Compute Node
NUMA Socket 1
NUMA Socket 2
P2_1
P1_1
S1
S2
P1_2
P2_2
C1
C2
C3
C4
C5
C6
(a). CFS Process Placement
(b). UniviStor Process Placement
P1_3
P1_4
idle
(c). Static Placement under Oversubscription
(d). Adaptive Placement with Migration under Oversubscription

Fig. 4: Interference-aware resource scheduling.

are served by UniviStor server processes S1 and S2. CFS is agnostic about which process belongs to which application and placing processes without such awareness may result in two issues. First, P2_1 and P1_1 are stacked together on the same core. This stacked placement not only incurs interference between P2_1 and P1_1 causing context switches, but also under-utilize the memory and network bandwidth provisioned by the other cores (e.g., C6). Second, without this awareness, processes from the same application may be placed in the same NUMA socket (e.g., S1 and S2 in Fig. 4(a)). Consequently, they can only use the CPU/memory/network resources from one NUMA socket.

As UniviStor servers are deployed across all the compute nodes allocated to each individual job, it has the knowledge of how many processes of each parallel program (including the server itself) are co-located on the same node. Using this insight, UniviStor servers evenly spread the processes of each program across all the NUMA sockets. In Fig. 4(b), processes from all the three programs (P_1_i, P_2_j, and S_k) are spread across cores on the two NUMA sockets. In this way, each process can use all the NUMA sockets without interfering with each other. In case a program's process count on a compute node is not divisible by the number of NUMA sockets, the remainder cores are assigned to the less loaded NUMA socket to balance NUMA resource utilization.

This strategy causes an issue when the total process count of the server and application programs exceeds the available CPU cores. A natural solution is to assign the additional processes to the existing cores already allocated to the same program. For instance, in Fig. 4(c), P1_3 and P1_4 are assigned on C1 and C4, respectively. This approach can potentially under-utilize other cores, since programs may stay at a distinct state (e.g., busy/idle/exit), and have differentiated demands for cores. In Fig. 4, S1 and S2 are both idle. This idle state is common for the typical scientific simulations: their life cycles alternate between computation and checkpoint phases; UniviStor servers are active only when an application finishes one or more rounds of computations, checkpoint data to the distributed DRAM layer or burst buffer, followed by the close operation that triggers server-side flush operation. To utilize the resources, UniviStor assigns the additional processes in a state-aware manner as shown in Fig. 4(d). When server-side flush is not triggered, application processes can utilize the server cores for computation (e.g. P1_3 and P1_4 use C3 and C6, respectively). When data flush is triggered, client processes on the server cores are migrated to other cores to let server program quickly complete. In Fig. 4(d), P1_3 and P1_4 can be migrated to C1 and C4 respectively. When the application exits, UniviStor server reassigns its cores to the existing processes based on the rules depicted in Fig. 4(b). This adaptive migration use all the cores efficiently under over-subscription scenario.

D. Adaptive data striping for fast data flush

UniviStor's server-side asynchronous flush operation allows applications to continue their computation without waiting for writing the data to PFS. The benefit of asynchronous writes heavily depends on UniviStor servers' flush bandwidth. A slow flush operation may lead to servers constantly competing with their client applications for shared resources (CPUs, memory, and network).

A simple and widely used approach to increase write bandwidth to PFS is to stripe (i.e., split) each shared file across all available storage devices (e.g., Object Storage Target (OST) in Lustre file system) and let each server write a contiguous file range evenly partitioned across the servers. However, this approach has limitations. For instance, striping each file across all storage units has no advantage when only a *small* number of servers are flushing data, because each server has to contact all storage units, and the additional synchronization overhead diminishes the benefits of a large stripe count [28], [29]. In contrast, a *large* number of servers concurrently flushing to all storage units causes load imbalance: the write requests are randomly directed to storage units. In any given time period, some storage units can receive many more write requests than others. This unbalanced load results in bandwidth under-utilization [30].

In order to address these issues, we have designed an adaptive data striping approach. The core idea is to adjust the striping pattern of files dynamically in two cases: 1) When the server count is smaller than the available storage units on PFS, UniviStor maximizes each server's flush bandwidth by striping its contiguous file range on a *distinct* set of storage units; 2) when the server count is larger, it *balances* concurrent flushing servers on each storage unit.

A key question for the first case is deciding on the number of distinct storage units each servers' file range is striped across (C_{per_server}). UniviStor calculates C_{per_server} by Equation (Eq.) 2.

$$C_{per\_server} = min(C_{max\_units}/C_{servers}, \alpha) \qquad (2)$$

In Eq. 2, C_{max_units} is the total storage unit count in the PFS and $C_{servers}$ is the server count. Our goal is to set C_{per_server} as large as possible given that it does not exceed α, α represents the minimum storage unit count that saturates a server's write bandwidth. The stripe size and count of a shared file based on C_{per_server} is derived in Eq. 3 and Eq. 4, respectively. In Eq. 3, S_{file} is the file size and S_{max} is the maximum allowable stripe size in the system.

$$S_{stripe} = min(S_{file}/(C_{servers} \times C_{per\_server}), S_{max}) \quad (3)$$

$$C_{stripe} = min(S_{file}/S_{stripe}, C_{max\_units}) \quad (4)$$

The second goal is balancing the workload when the number of flushing servers is larger than the maximum number of storage units. As the servers need to overlap their flush on the storage units. In this case, one potential approach to evenly distribute the per storage unit's workload is by configuring the stripe size following Eq. 5.

$$S_{stripe} = S_{file}/C_{servers} \quad (5)$$

In Eq. 5, the file range of each server is striped to one storage unit, and all storage units are assigned to servers in a round robin manner. This approach balances the number of flushing servers per storage unit when $C_{servers}$ is divisible by C_{max_units}. However, load balancing issue still remains otherwise. For instance, assume a Lustre file system has 248 OSTs. Using 512 flushing servers, 16 OSTs (512%248) have to sustain the flushing workload from one additional process and become potential stragglers. To resolve this issue, we adjust $C_{servers}$ to $C_{dum\_servers}$ (Eq. 6).

$$C_{dum\_servers} = \lceil C_{servers}/C_{max\_units} \rceil \times C_{max\_units} \quad (6)$$

It can be perceived that $C_{dum\_servers} \geq C_{servers}$, resulting in a smaller stripe size than that in Eq. 5. In our example, $C_{dum\_servers}$ is set to 724 instead of 512. Applying this larger value to Eq 5 results in a smaller stripe size that amortizes the workloads across all OSTs.

E. Lightweight workflow management

Uncoordinated redirection of application I/O requests to the UniviStor servers may lead to reading stale data. For example, an analysis application can read incomplete or stale data when it reads the file being written by a simulation application. Without coordination, the read requests have to wait until the writing process completes, or the application developers have to resolve data dependency by writing their own code in order to run these applications concurrently. Similarly, an application may overwrite a file that is being read/written by another application. To avoid conflicting accesses and to enable in-situ/in-transit analysis on distributed and hierarchical storage layers, we designed a lightweight workflow management scheme in UniviStor that coordinates applications with data dependencies. Users can enable this feature optionally by setting an environment variable ENABLE_WORKFLOW. An application attempting to acquire a lock on a file for reading has to wait until another writing application release it. Similarly, a writer has to wait until another reader/writer releases the lock on the shared file. UniviStor attaches the lock acquire/release operations to MPI_File_open/MPI_File_close, instead of forcing applications to use separate lock functions.

Internally, lock acquire/release is enforced by monitoring/updating a shared state file located on a file system (e.g. PFS): A writing/reading application locks a file by updating this file's record in the state file to WRITING / READING state; it releases the lock by updating its state to WRITE_DONE / READ_DONE. With these operations, a writing application operating on a file needs to wait if it is in WRITING / READING state, and update its state to WRITE_DONE upon completing the writes. Similarly, a reading application on a file needs to wait if the file is in WRITING state, and update its state to READ_DONE upon read completion. In addition, FLUSHING and FLUSH_DONE states are defined to avoid the conflicting accesses when a write application is writing a file currently being flushed by the servers to PFS.

A key design consideration is monitoring/updating the state file for locking. In particular, an approach that involves all the processes of the writing/reading applications concurrently operating on the state file can largely offset UniviStor's performance benefit. Instead, UniviStor piggybacks lock acquire/release operations with MPI_File_open/MPI_File_close operations in the MPI-IO layer, and only allows the root process to operate on the state file. When an application opens a file in a write-only/read-only mode, it attempts to acquire the write/read lock. When an application closes a file in write-only/read-only mode, it releases the write/read lock. This design choice is based on the fact that MPI_File_open/MPI_File_close are collective operations, attaching the locking mechanism to these functions avoids additional synchronization overhead.

F. Implementation

We have implemented UniviStor client in MPICH 3.3 [31], where the UniviStor server is a separate MPI program launched across all the compute nodes allocated to a job. For transparent support of the standard parallel I/O libraries, such as MPI-IO, HDF5, and netCDF that are generally stacked on top of POSIX, a typical solution is to directly intercept the underlying POSIX function calls. However, this approach requires an extensive implementation of the full list of POSIX functions and demands that the implementation complies with the POSIX semantics, whose consistency model is widely considered as the key performance limiting factor.

In order to make our system efficient, portable, and transparent, we have developed an UniviStor I/O driver for MPI-IO's Abstract-Device Interface (ADIO) [32], and implemented UniviStor within this layer. ADIO allows file system developers to implement their own file system feature for MPI-IO while exposing to applications the same MPI-IO interface. As MPI-IO is the underlying library for other high-level parallel I/O libraries (e.g., HDF5), this approach makes our system readily extensible to these libraries.

Our I/O driver implementation is optimized based on MPI-IO semantics. For instance, when a shared file is opened/closed by all the processes, all the processes have to send the same metadata requests (e.g., file attributes) to the same UniviStor server (determined by the file name hash). However, this all-to-one operation is not a scalable option. In our ADIO implementation, when all processes open/close a shared file using MPI-IO, only the root (MPI rank #0) process operates on the metadata and broadcasts the results to other processes. We have also provided an option for users to optimize the HDF5 operations. When processes open/close a shared file using HDF5 without a collective optimization, all processes need to read/write the metadata region stored on the same UniviStor server. Instead, when the user enables HDF5 optimization, our ADIO layer is able to detect HDF5 open/close calls and only the root process operates on the metadata region and broadcasts the result to all other processes.

III. Evaluation

To evaluate UniviStor, we first used micro-benchmark workloads using various optimizations (§III-B). We then used I/O workloads from real scientific applications to compare UniviStor with the state-of-the-art storage management solutions (§III-C). We have also evaluated UniviStor's support for scientific workflows with reading and writing data on multiple storage layers (§III-D).

A. Experimental setup

We ran all our experiments on Cori, a Cray XC40 system located at the National Energy Research Scientific Computing Center (NERSC), consisting of 2388 Intel Xeon "Haswell" compute nodes and 9688 Intel Xeon Phi "Knight's Landing" nodes. Our experiments used the "Haswell" partition, where each node has 32 cores spread across two NUMA sockets and 128 GB DDR4 2133MHz DRAM. All nodes are connected to a Lustre file system with 248 object storage targets (OSTs). Additionally, this system has been deployed with a layer of the shared burst buffer. The burst buffer is allocated to individual jobs based on the request made in a job script.

Comparisons: In this evaluation, we compare UniviStor with Data Elevator [14] and Lustre [33]. Data Elevator is a software library to use shared burst buffer as a caching layer before writing data to PFS. It relies on DataWarp [4] to manage the burst buffer and Lustre as PFS. In comparison, besides the shared burst buffer and PFS, UniviStor unifies the distributed storage devices (i.e., DRAM in this evaluation) on all compute nodes. UniviStor also provides a "file-per-process" format (§II-B1) to store the cached data. Lustre is a traditional and the most popular disk-based parallel file system for HPC, but it does not support data caching on burst buffers. Applications can only use Lustre to write data from local DRAM to the file system. The comparisons between UniviStor with Lustre shows the benefits of managing data on both the DRAM and a shared burst buffer.

I/O workloads: We have used both micro-benchmarks and representative I/O workloads from real scientific applications. The micro-benchmarks are from the HDF5 source code [34], where each process creates a shared HDF5 file and writes/reads an independent but overall contiguous block of data. The scientific I/O workloads, including VPIC-IO [35] and BD-CATS-IO [36], are I/O kernels of a large-scale space weather plasma simulation code and a corresponding data analysis code. In VPIC-IO, each MPI process writes data related to eight million particles, and each particle has eight floating point properties with a total size of 32 bytes. The total size of output data is $n \times 8 \times 2^{20} \times 32$, where n is the number of MPI processes. BD-CATS-IO implements reading properties from the datasets similar to that produced by VPIC, for a parallel clustering algorithm to identify the irregularly shaped clusters from the particles. Our tests read all eight properties of all particles, similar to the BD-CATS analysis.

Performance metrics: We measured the time required to open, write, read, and close a file. We define I/O rate as the ratio of the size of data read/written to the I/O time. We placed two UniviStor servers on each compute node to exploit UniviStor's NUMA benefits. We placed the same number of Data Elevator servers, i.e., 2 on each compute node. We scale the application from 64 to 8192 processes with $2\times$ increments and run each test at least three times and report the best performing results.

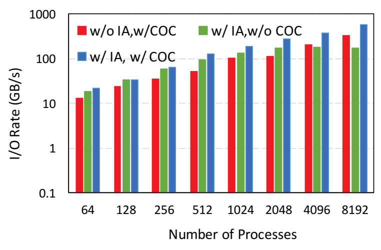
B. Evaluation with micro-benchmarks

In this section, we evaluate the optimizations of UniviStor for reading, writing, and flushing with I/O workloads on both distributed and hierarchical storage, and compare with state-of-the-art solutions.

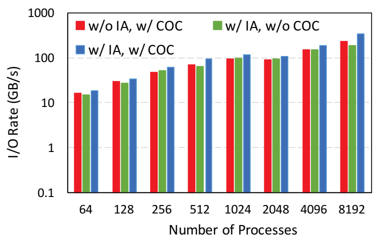
I/O with interference-aware scheduling (IA) and collective file open/close (COC). We report the performance of writing 256MB data per process to UniviStor's distributed DRAM space in Fig. 5a. We can observe that each feature has a distinct benefit, as the performance drops evidently when either feature is disabled. IA avoids inter-process interference and efficiently utilizes the on-node resources, and COC transforms the all-to-one/one-to-all communication to one-to-one communication between the root client and the server. For the read performance (in Fig. 5b), we observed similar benefits. Overall, as the number of processes varies, the combined improvement of IA and COC is from $1.45\times$ to $2.5\times$ ($1.9\times$ on average), $1.1\times$ to $3.5\times$ ($1.6\times$ on average) speedup compared with when either is disabled, respectively, for the writes; and 1.13x to $1.5\times$ ($1.25\times$ on average), $1.15\times$ to $1.8\times$ ($1.3\times$ on average) speedup, respectively, for the reads.

Server-side data flush with adaptive data striping (ADPT) and interference-aware resource scheduling (IA). In this test, we explore the impact of IA and ADPT (see Section II-D) when data are flushed from UniviStor's distributed DRAM space to the Lustre PFS. Fig. 5c reports the flushing I/O rate. When IA is enabled, the co-located client processes are migrated away from the flushing servers during the flush, and moved back after the flush, allowing the servers to flush without interference from the co-located clients. When ADPT is enabled, each flushing server avoids the synchronization overhead with the OSTs and balances the per OST workload on Lustre. The results suggest that enabling both IA and ADPT can improve the performance by $1.9\times$ to $2.7\times$ ($2.3\times$ on average).

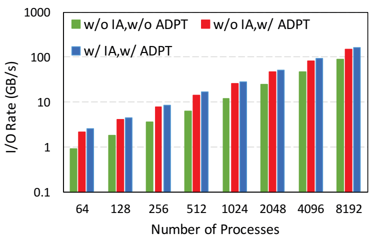
Comparing UniviStor with Data Elevator and Lustre using micro-benchmarks. Fig. 6a compares UniviStor's overall write performance with Data Elevator and Lustre. In this comparison, we focus on investigating UniviStor's performance on



(a) Write

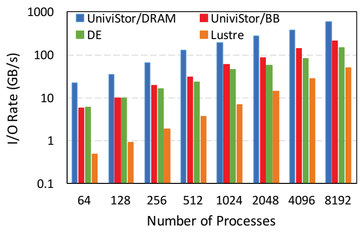


(b) Read

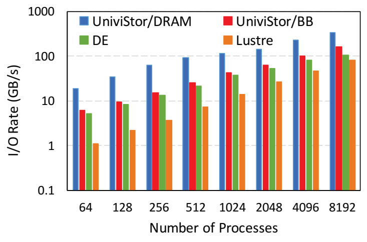


(c) Flush

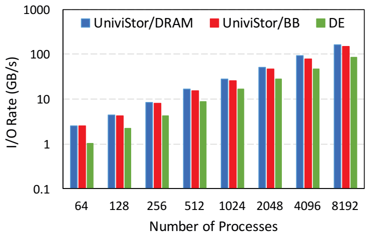
Fig. 5: Performance of writing, reading, and flushing data with Interference-Aware (IA) resource scheduling, Collective Open/Close (COC), or ADaPTive data striping (ADPT). The y-axis is logarithmic scale.



(a) Write



(b) Read



(c) Flush

Fig. 6: Comparing UniviStor with Data Elevator and Lustre with micro-benchmarks. The y-axis is logarithmic scale.

each storage layer with all optimizations enabled. We observe that UniviStor using both DRAM and burst buffer outperform Data Elevator and Lustre. UniviStor/DRAM performs the best due to the raw bandwidth benefit from UniviStor's distributed DRAM. The performance advantage of UniviStor/BB over Data Elevator is because UniviStor reorganizes processes' writes on burst buffer using the "file-per-process" format, whereas Data Elevator lays out processes' data in one shared HDF5 file. Overall, UniviStor/DRAM and UniviStor/BB outperform Data Elevator by 3.7× to 5.6× (4.3× on average) and 1.2× to 1.7× (1.3× on average), respectively, as the number of processes varies between 64 and 8192. Compared with Lustre that stores data on the disk-based PFS, UniviStor/DRAM and UniviStor/BB deliver up to 46× and 12× performance improvement, respectively. We observe the same pattern for the read performance, as shown in Fig. 6b. Overall, UniviStor/DRAM and UniviStor/BB outperform Data Elevator by 2.7× to 4.5× (3.6× on average) and 1.15× to 1.6× (1.2× on average) for read, respectively. They deliver up to 16.8× and 5.4× speedup over Lustre, respectively.

UniviStor supports flushing from DRAM and burst buffers to the persistent storage (e.g. Lustre). Data Elevator only supports flushing from burst buffers to Lustre. Hence, we track the time for UniviStor in two different cases: UniviStor/DRAM and UniviStor/BB. They refer to I/O rate of flushing data from DRAM to Lustre and burst buffer to Lustre. Fig. 6c compares UniviStor's flush I/O rate with Data Elevator, we can see that the I/O rate of UniviStor/BB are higher than Data Elevator since UniviStor balances the workload on Lustre OSTs and avoids servers' synchronization overhead with

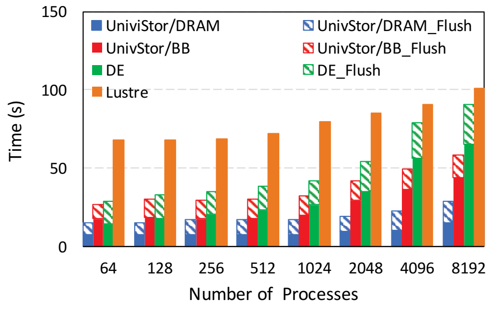


Fig. 7: Total I/O time of writing 5-time-step VPIC-IO data.

OSTs using ADPT discussed in Section II-D. Meanwhile, we observed even higher performance in UniviStor/DRAM, due to the faster DRAM bandwidth than burst buffer. Overall, UniviStor/DRAM and UniviStor/BB outperform Data Elevator by 1.8× to 2.5× (2× on average) and 1.6× to 2.5× (1.8× on average), respectively.

C. Evaluation with scientific I/O workloads

Scientific simulations such as VPIC [35] typically progress in time steps. After one or more time steps of computations, all processes concurrently checkpoint data to the storage system. We use the VPIC-IO kernel to evaluate UniviStor's support for such I/O workload. In VPIC-IO, each process writes eight variables with a total size of 256MB in each time step. We run VPIC-IO with 5 time steps and 10 time steps. Based on our current hardware configuration, UniviStor's distributed DRAM space is insufficient for containing data pertaining to 10 time steps. The additional data has to be spilled to a storage layer, such as a burst buffer. To emulate the computation behavior,

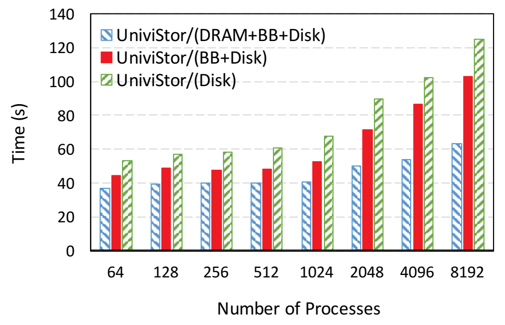


Fig. 8: Total I/O time of writing 10-time-step VPIC-IO data with UniviStor using different storage layers.

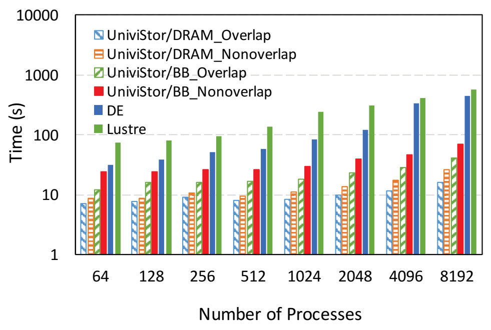


Fig. 9: Total time of finishing the workflow consisting of 5-time-step VPIC-IO and BD-CATS-IO. Y-axis is logarithmic scale.

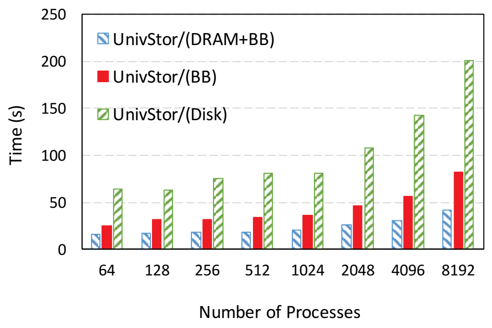


Fig. 10: Elapsed time of the workflow consisting of a 10 time steps of VPIC-IO writes and BD-CATS-IO reads.

we manually add a sleep interval of 60 seconds between checkpoints. Both UniviStor and Data Elevator overlap the server-side flush operation with the sleep time, by temporarily caching data in DRAM or burst buffer, and letting servers asynchronously flush the data to the Lustre file system. Therefore, I/O time of both UniviStor and Data Elevator include the time for writing to DRAM/burst buffer layer, as well as flushing the last timestep data (labeled with "Flush" in suffix in the plots). The I/O time for Lustre represents only the time for writing to the Lustre file system for all the time steps.

VPIC-IO with 5 time steps on a single layer of storage space. In Fig. 7, we compare the I/O time with UniviStor, Data Elevator (DE), and Lustre to store the data generated by VPIC-IO. We configure UniviStor to write its data to DRAM (UniviStor/DRAM) and to burst buffer (UniviStor/BB). As expected, writing data to DRAM is the fastest. On the other hand, the performance of UniviStor/BB is almost equal to that of DE at smaller scale, but gradually outperforms DE as more processes are involved. This is because UniviStor/BB's "file-per-process" format mitigates the contention issue faced by DE at larger scales. It is also observable that UniviStor/BB's flush is faster than that of DE, due to the flush optimization introduced in UniviStor (§ II-D). Overall, UniviStor/DRAM and UniviStor/BB are 1.9× to 3.1× (2.5× on average) and 1.1× to 1.6× (1.3× on average) faster than that of DE.

VPIC-IO with 10 time steps on multiple layers of storage. In this test case, the accumulated VPIC-IO data does not fit in the DRAM and UniviStor has to spill roughly half of the data to burst buffer. We label to this case as (DRAM+BB+Disk) in Fig. 8, where "Disk" means the time for flushing the last time step data to the file system. For comparison, we measure the time of caching data entirely in the burst buffer (BB+Disk) and Lustre file system (Disk). It can be observed that UniviStor/(DRAM+BB+Disk) writes much faster than UniviStor/(BB+Disk) and UniviStor/(Disk), with 1.2× to 1.6× (1.4× on average) speedup over UniviStor/(BB+Disk), and 1.4× to 2× (1.7× on average) speedup over UniviStor/(Disk), respectively, because UniviStor benefits from both the DRAM and burst buffer bandwidth. These tests also demonstrate the benefits of UniviStor's data management on distributed and hierarchical storage.

D. *Support for scientific workflows*

Scientific applications often involve workflows, where data producers and consumers share data. To evaluate UniviStor's support for scientific workflows, as introduced in § II-E, we use BD-CATS-IO [36] to read data produced by VPIC-IO. We configure both programs to run 5 time steps and 10 time steps. Similar to the experiment in § III-C, data produced in 10 time steps does not fit entirely in the DRAM layer and UniviStor has to spill the data to burst buffer. We evaluate UniviStor's workflow management by comparing two modes: BD-CATS-IO and VPIC-IO running concurrently orchestrated by UniviStor's workflow management ("overlap" mode); and BD-CATS-IO running after VPIC-IO finishes all its time steps ("nonoverlap" mode). We evaluate UniviStor's acceleration of data movement across storage layers by comparing UniviStor's non-overlap mode with Data Elevator and Lustre using the same execution sequence, i.e., BD-CATS-IO starts after VPIC-IO finishes. We configure VPIC-IO and BD-CATS-IO to use half the number of processes each, and measure the elapsed time as the interval between the start of VPIC-IO and the end of BD-CATS-IO. We show in Fig. 9 the results of a 5-time-step run. UniviStor/DRAM and UniviStor/BB refer to the scenario where VPIC-IO data are written to the DRAM and to the burst buffer layers, respectively. We can observe that with the workflow management, performance of UniviStor/DRAM_Overlap and UniviStor/BB_Overlap are both faster than UniviStor/DRAM_Nonoverlap and UniviStor/BB _Nonoverlap, accounting for 1.2× to 1.7× (1.3× on average) and 1.5× to 2× (1.7× on average) performance improvement, respectively. In addition, UniviStor/DRAM_Nonoverlap and UniviStor/BB_Nonoverlap still demonstrate 3.5× to 17× (9× on average) and 1.3× to 7.2× (3.4× on average) performance improvement over DE, respectively. The main reason for the performance benefit of UniviStor/DRAM_Nonoverlap comes from UniviStor's fast data movement on distributed DRAM layer. Moreover, the benefit on UniviStor/BB_Nonoverlap is because UniviStor's "file-per-process" data transformation on BB substantially accelerates both VPIC-IO's write operations and BD-CATS-IO's read operations.

We demonstrate the benefit of a unified view of storage in Fig. 10 by comparing the elapsed time for supporting 10 time-step workflow. In UniviStor/(DRAM+BB), 10-time-step data is spread across the distributed DRAM layer and the burst buffer. For comparison, we also measure the time when all data are placed on burst buffer (UniviStor/(BB)) and the Lustre file system (UniviStor/(Disk)). We noticed that placing data on both DRAM and BB can achieve 1.5× to 2× (1.8× on average) speedup over placing the data only on BB and 4× to 4.8× (4.3× on average) over placing data on Lustre.

IV. Related Work

HPC storage hierarchy evolution has fostered various data management solutions for each layer. Traditional efforts primarily manage data on the hard-disk based PFS, such as Lustre [33], OrangeFS [37], GPFS [38], etc. The striping-based data placement adopted by most PFSs cannot distinguish the devices across layers, and I/O performance of PFS varies significantly due to locking and interference [39], [15], [40].

There are several recent burst buffer management software developments. Cray DataWarp [4] and DDN IME [5] are two vendor solutions for shared burst buffers. DataWarp stripes a file across burst buffer nodes similar to conventional PFSs. IME adaptively places data on burst buffer nodes based on their loads. BeeOND [41] and BurstFS [23] are open-source software for node-local SSDs. BeeGFS On Demand (BeeOND) creates a file system on demand per job and stripes each file across multiple nodes similar to DataWarp. BurstFS directs each process's writes to its local SSD for scalable write bandwidth. PLFS [25] has the feature to direct processes' IO to either node-local or shared burst buffer. In contrast, UniviStor is designed to unify various node-local and shared storage layers.

A few burst buffer libraries have transparent caching feature that redirects writes to the PFS to burst buffer transparently, and asynchronously flush data to the PFS, such as DataWarp, Data Elevator [14] and Spectrum Scale [13]. By comparison, UniviStor can cache data on any of the available layers, including DRAM, and its data flush operation is highly optimized.

In managing distributed DRAM, solutions such as OpenShmem [2], UPC [1], DataSpaces [3] expose to a user library-specific interfaces for globally addressing the distributed memory. In contrast, UniviStor is designed for a large number of HPC applications using parallel I/O (e.g. MPI-IO, HDF5). In addition, UniviStor supports a lightweight workflow management for in-situ/in-transit analysis on DRAM and burst buffer.

There are a few efforts for managing data on multiple storage layers. CRUISE [42] is a checkpoint file system that writes either to node-local DRAM/SSD or to PFS. SSUDP [43] and BurstMem [44], [45] are burst buffer systems that write to shared burst buffer or to PFS. These efforts generally move data between specific storage layers. In contrast, UniviStor supports all available layers that are either node-local or shared across the compute partition. DAOS [9] is an object-based file system solution being developed for managing data objects in a hierarchical storage layers using transactions. DAOS can be accessed with HDF5 API, and its current deployment on HPC systems requires significant storage infrastructure modifications. Hermes [10] is recent I/O buffering system developed to manage multiple storage layers. It offers three policies to navigate data placement across layers, and introduces three novel techniques to perform memory, metadata, and communication management. In contrast, UniviStor is a transparent service and does not need users to know the behavior of their applications in advance. While Hermes' service is implemented as a library linked to the individual application, UniviStor is an independent parallel program providing data sharing service for multiple coupled applications.

V. Conclusions and Future Work

With the goal of providing an integrated storage subsystem on supercomputing machines with multiple layers of storage, we have designed and implemented UniviStor. Applications using MPI-IO and HDF5 libraries for performing I/O can take advantage of UniviStor without any source code changes. Our system accelerates I/O of these applications using efficient data placement and movement across hierarchical storage layers. Our evaluation on a leadership-class computing system and comparison to the state-of-the-art data management solutions demonstrates that UniviStor achieves efficient scientific application I/O on upcoming exascale storage architectures without burdening the users on modifying existing applications. In particular, it outperforms Data Elevator and Lustre by up to 17× and 46×, respectively. We are exploring various enhancements to UniviStor as future work, including adding resilience to data in volatile storage layers and adaptive and proactive placement of data based on data usage patterns.

Acknowledgment

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 (Project: Proactive Data Containers, 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] T. El-Ghazawi and L. Smith, "UPC: Unified Parallel C," in *SC*. ACM, 2006, p. 27.

[2] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS Community," in *PGAS*. ACM, 2010, p. 2.

[3] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[4] Cray, "Cray DataWarp Web Page," http://www.cray.com/products/storage/datawarp, 2017.

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

[6] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*. IEEE, 1999, pp. 182–189.

[7] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel NetCDF: A High-Performance Scientific I/O Interface," in *SC*. IEEE, 2003, pp. 39–39.

[8] T. Wang, K. Vasko, Z. Liu, H. Chen, and W. Yu, "BPAR: A Bundle-Based Parallel Aggregation Framework for Decoupled I/O Execution," in *DISCS*. IEEE Press, 2014, pp. 25–32.

[9] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and Friends: A Proposal for An Exascale Storage System," in *SC*. IEEE Press, 2016, p. 50.

[10] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System," in *HPDC*. ACM, 2018, pp. 219–230.

[11] NERSC. DataWarp Usage. Http://www.nersc.gov/users/computational-systems/cori/burst-buffer/example-batch-scripts/.

[12] Cray, "Burst Buffers," http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/, 2017.

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

[14] 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*. IEEE, 2016, pp. 152–161.

[15] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *SC*. IEEE, 2010, pp. 1–12.

[16] H. Jin, T. Ke, Y. Chen, and X.-H. Sun, "Checkpointing Orchestration: Toward a Scalable HPC Fault-Tolerant Environment," in *CCGRID*. IEEE, 2012, pp. 276–283.

[17] T. Wang, K. Vasko, Z. Liu, H. Chen, and W. Yu, "Enhance Parallel Input/Output with Cross-Bundle Aggregation," *IJHPCA*, vol. 30, no. 2, pp. 241–256, 2016.

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

[19] Z. Liu, B. Wang, T. Wang, Y. Tian, C. Xu, Y. Wang, W. Yu, C. A. Cruz, S. Zhou, T. Clune *et al.*, "Profiling and Improving I/O Performance of a Large-Scale Climate Scientific Application," in *ICCCN*. IEEE, 2013, pp. 1–7.

[20] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics," in *CCGRID*. IEEE, 2014, pp. 246–255.

[21] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "Fusionfs: Toward Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems," in *BigData*. IEEE, 2014, pp. 61–70.

[22] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu, and R. Warren, "Towards Scalable and Asynchronous Object-Centric Data Management for HPC," in *CCGRID*. IEEE, 2018.

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

[24] 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*. IEEE, 2017, pp. 1174–1183.

[25] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *SC*. IEEE, 2009, pp. 1–12.

[26] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia *et al.*, "Accelerating Science with the NERSC Burst Buffer Early User Program," *CUG*, 2016.

[27] C. S. Pabla, "Completely Fair Scheduler," *Linux Journal*, vol. 2009, no. 184, p. 4, 2009.

[28] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *SC*. IEEE, 2011, pp. 1–11.

[29] W. Yu, J. Vetter, R. S. Canon, and S. Jiang, "Exploiting Lustre File Joining for Effective Collective IO," in *CCGRID*. IEEE, 2007, pp. 267–274.

[30] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "TRIO: Burst Buffer Based I/O Orchestration," in *CLUSTER*. IEEE, 2015, pp. 194–203.

[31] ANL, "MPICH," https://www.mpich.org/, 2018.

[32] R. Thakur, W. Gropp, and E. Lusk, "An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces," in *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the*. IEEE, 1996, pp. 180–187.

[33] P. J. Braam and R. Zahir, "Lustre: A Scalable, High Performance File System," *Cluster File Systems, Inc*, 2002.

[34] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and Its Applications," in *EDBT/ICDT*, 2011, pp. 36–47.

[35] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughter, 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*, 2012, pp. 59:1–59:12.

[36] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "BD-CATS: Big Data Clustering at Trillion Particle Scale," in *SC*, 2015.

[37] M. M. Bonnie, B. Ligon, M. Marshall, W. Ligon, N. Mills, E. Q. S. Sampson, S. Yang, and B. Wilson, "OrangeFS: Advancing PVFS," *FAST poster session*, 2011.

[38] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.

[39] W.-k. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," in *SC*. IEEE Press, 2008, p. 3.

[40] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *IPDPS*. IEEE, 2016, pp. 750–759.

[41] J. Heichler, "An Introduction to BeeGFS," 2014.

[42] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda, "A 1 PB/s File System to Checkpoint Three Million MPI Tasks," in *HPDC*. ACM, 2013, pp. 143–154.

[43] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. Chen, "SSDUP: A Traffic-Aware SSD Burst Buffer for HPC Systems," in *ICS*. ACM, 2017, p. 27.

[44] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "Burstmem: A High-Performance Burst Buffer System for Scientific Applications," in *BigData*. IEEE, 2014, pp. 71–79.

[45] T. Wang, W. Yu, S. Oral, B. W. Settlemyer, and S. Atchley, "An Efficient Distributed Burst Buffer for Linux," in *LUG*, 2014.