HCompress: Hierarchical Data Compression for Multi-Tiered Storage Environments

Hariharan Devarajan, Anthony Kougkas, Luke Logan, and Xian-He Sun
Department of Computer Science, Illinois Institute of Technology, Chicago, IL
hdevarajan@hawk.iit.edu, akougkas@iit.edu, llogan@hawk.iit.edu, sun@iit.edu

***Abstract*—Modern scientific applications read and write massive amounts of data through simulations, observations, and analysis. These applications spend the majority of their runtime in performing I/O. HPC storage solutions include fast node-local and shared storage resources to elevate applications from this bottleneck. Moreover, several middleware libraries (e.g., Hermes) are proposed to move data between these tiers transparently. Data reduction is another technique that reduces the amount of data produced and, hence, improve I/O performance. These two technologies, if used together, can benefit from each other. The effectiveness of data compression can be enhanced by selecting different compression algorithms according to the characteristics of the different tiers, and the multi-tiered hierarchy can benefit from extra capacity. In this paper, we design and implement HCompress, a hierarchical data compression library that can improve the application's performance by harmoniously leveraging both multi-tiered storage and data compression. We have developed a novel compression selection algorithm that facilitates the optimal matching of compression libraries to the tiered storage. Our evaluation shows that HCompress can improve scientific application's performance by 7x when compared to other state-of-the-art tiered storage solutions.**

***Index Terms*—hierarchical, multi-tiered, data compression, data-reduction, dynamic choice, workflow priorities, library**

I. Introduction

Modern scientific applications, spanning from complex simulations to high-performance data analytics, generate, process, and store extremely large amounts of data [1]. The ability to process this explosion of data is now driving scientific discovery more than ever [2]. These applications, widely known as data-intensive computing, often spend a significant amount of time in performing I/O [3]. In High-Performance Computing (HPC), the underlying data are represented by files that, typically, are stored in disk-based parallel file systems (PFS). However, the evolution of processing technology (i.e., CPU, RAM) has far exceeded the storage capabilities which has led to a performance gap between the compute and storage resources. This phenomenon is known as the *I/O bottleneck* [4], and causes performance degradation, complex deployment models, and limited scalability.

Several middleware software solutions have been introduced to tackle this I/O bottleneck problem. For instance, data access optimizations such as data buffering and caching, data prefetching, and data aggregations hide the gap between compute and I/O by offloading the I/O cost to an intermediate temporary space (e.g., main memory, buffers, SSD, etc.,) so that application can continue the execution while data is moved asynchronously to a slower remote shared storage (e.g., PFS). Similarly, some system architectures include intermediate, specialized resources such as I/O forwarders, burst buffers, and data staging nodes that are closer to the compute nodes, and thus, offer lower data access latency and higher throughput, effectively masking the I/O gap. Finally, data reduction optimizations such as deduplication [5], filtering [6], subsampling [7], and compression [8] aim to reduce the data footprint, and thus, the cost of moving large amounts of data to/from the slower remote storage system. The effectiveness of all these solutions is directly proportional to the amount of available space in an Intermediate Temporary Storage (ITS) (i.e., scratch space) and its utilization. The more capacity that the ITS has, the more important the I/O utilization becomes. As the amount of RAM per core keeps decreasing [9], it is imperative, more than ever before, to maximize the resource and capacity utilization of ITS space.

There are two orthogonal approaches to achieve this goal: one that relies on the addition of new hardware that increases the available ITS capacity and another that opts to reduce the amount of data redirected to ITS. First, modern storage systems have introduced new specialized hardware technologies such as High-Bandwidth Memory (HBM), Non-Volatile RAM (NVRAM), and Solid-State Drives (SSD) by organizing them in a multi-tiered storage hierarchy [10] (i.e., higher tiers are faster but smaller in capacity). To overcome the added complexity in the storage stack, several middleware solutions such as Hermes [10] and Univistor [11] have been introduced to transparently manage these tiers. However, the performance of such multi-tiered solutions is determined by their ability to place more data in the upper tiers, which offer lower latency and higher bandwidth.

Second, there are several software-based solutions that aim to reduce the data volume placed on ITS. The most commonly used is data compression [12], [13], where CPU cycles are used to apply compression/decompression on the data before they are stored/retrieved to/from the disks, resulting in shorter I/O times caused by the smaller data footprint. Thus, there is a trade-off between the amount of time spent compressing data and the amount that I/O cost is reduced. This has led to a plethora of specialized compression libraries that optimizes data compression based on certain types of data, workloads, or domains [14]. However, the performance characteristics of different data compression solutions are often determined by their ability to balance the used resources effectively (CPU cycles - I/O device target). Thus, choosing the right compression tech-

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00064

nique for a given scenario and target environment is nontrivial.

While the above solutions have been proven to be effective at increasing the available ITS space, they work in isolation which can lead to resource under-utilization, poor flexibility, and inelastic configuration of ITS. In this study, we focus on the intersection of multi-tiered storage and data compression. In such an environment, we make the following observations: a) *Multi-tiered storage does not utilize data compression*. Existing software places data in the tiers without applying compression, which leads to under-utilization of expensive resources (e.g., NVMe). At best, some solutions may naively apply the same compression across all tiers which leads to a missed optimization opportunity since each tier may benefit from different compression algorithms. If data compression was efficiently utilized, it would increase the overall capacity of the tiers improving the effectiveness of I/O optimizations (e.g., more buffered or cached data, more aggregation space, etc.). b) *Data compression is not tailored for multi-tiered storage.* Compression/decompression often benefits from balancing the time spent in compression with the achieved reduction in I/O time. However, in multi-tiered environments, this trade-off can be applied per-tier increasing the optimization opportunity of this technique. For the same overall time budget, one could apply heavier compression on RAM than on NVMe SSD (as the medium is faster). For a given input, this creates a matching of the performance characteristics of different compression choices and the storage tiers.

In this paper, we present HCompress, a new hierarchical data compression engine for multi-tiered storage environments that maximizes the available ITS space while minimizing the overall time spent. It does so by using a multi-dimensional dynamic programming optimization algorithm to optimally match the following: a) incoming data attributes (e.g., data type, size), b) each tier's specifications (e.g., capacity, latency, throughput), and c) compression characteristics (e.g., compression/decompression speed and ratio). HCompress uses a cost-based model, enhanced with reinforcement learning, to predict the expected performance of each compression library and storage tier combination. HCompress can attach to existing multi-tiered software to enable transparent hierarchical data compression. The main contributions of this work are:

1) Demonstrating how the combination of data compression and multi-tiered storage can optimize ITS (III).
2) The design of HCompress, a hierarchical compression library for tiered storage environments (IV).
3) A novel compression selection algorithm which maps the compression libraries to the tiered storage (IV-F1).
4) Quantifying the benefits of a hierarchical compression engine for complex scientific workloads (V).

II. Background and Related Work

Scientific computing deals with overwhelming amounts of data [15] which increases the performance gap between computing speed and traditional storage systems [16]. To tackle this I/O bottleneck problem, several middleware software solutions have been introduced to improve data access to/from storage resources. Data buffering [17] boosts the write-performance of an application by storing data into a fast intermediate device (e.g., RAM) and, then, asynchronously flushing it to the underlying storage. Data aggregation [18] gathers data from various sources into a temporary space (e.g., RAM) and collectively writes it to the underlying storage system in an efficient manner. Data caching [19] boosts an application's I/O performance by keeping frequently accessed data in a cache so that, when the same data is needed next time, it could be quickly retrieved. Data Prefetching [20] reads data into a temporary prefetching cache ahead of a read operation to mask the disk access latency. Data staging [21] pre-loads data in a set of specialized staging resources before the application starts to optimize data access. All these I/O optimizations critically depend on the availability of the ITS space to maximize their performance benefits. The community has proposed two promising, but highly orthogonal, approaches to improve the capacity and utilization of the ITS.

A. Multi-Tiered storage hierarchy

Latest supercomputer designs include new hardware technologies in an effort to mask the I/O gap between compute nodes and the remote shared PFS. To handle the high data production rate, compute nodes are now equipped with fast node-local devices such as Non-Volatile Memories (NVM) [22] including Solid State Drives (SSDs), Phase-change Memory (PCM) [23], Resistive-RAM (ReRAM) [24], and High-Bandwidth Memory (HBM) [25]. In addition, some system designs include a flash-only shared burst buffer tier [26]. For example, National Energy Research Scientific Computing Center (NERSC) Cori machine [27] uses CRAYs Datawarp technology [28]. Trinity [29] supercomputer, at the Los Alamos National Laboratory, uses a burst buffer installation of 3.7 PB capacity and 3.3 TB/s bandwidth. Such burst buffer deployments can offer higher throughput and lower latency due to their proximity to compute nodes and the more capable storage hardware [30]. However, traditional file systems are designed to manage a single tier of homogeneous storage devices [31], and, thus are not ready to handle the added complexity in the storage stack. Each of the added tiers of the memory and storage hierarchy are independent systems which increases the complexity for the end users [32]. To overcome this, several middleware solutions such as Hermes [10], Univistor [11] and Proactive Data Containers [33] have been introduced to transparently manage these layers. However, the capacity of the added tiers is orders of magnitude smaller than the PFS and none of the above multi-tiered software utilizes any data reduction techniques that could significantly increase the tier utilization. Data reduction optimizations are largely unexplored in these multi-tiered environments.

B. Data Compression Techniques

Complex scientific workflows frequently exchange critical data between its various components [34]. Some common data interactions include inter-process communication [35], archival storage [36], data aggregation [37], etc. To optimize these interactions, applications often use data compression [38]

to reduce the amount of data transmitted between these components or to external storage. For instance, applications such as AstroPortal [39], Community Earth System Model [40], and Particle Physics simulations [41] utilize compression to reduce the cost of data movement within the application. Similarly, many HPC applications [38], [42] perform data compression to reduce the amount of intermediate data produced in the staging servers. All these applications showcase the different compression needs of these workflows. Common compression algorithms leverage the nature of data to encode them more efficiently. For instance, Snappy [43] was built to work best on textual data, whereas quickLZ [44] works best for integer data. Authors of [45] have shown that this approach could lead to sub-optimal performance. In our previous work [14], we explored the benefit of optimizing compression based on the desired application priorities. Additionally, data compression could be further optimized since most of these HPC facilities are now equipped with multi-tiered storage. For instance, based on which tier of hierarchy the data is transmitted to, a suitable compression algorithm could be chosen to match the performance characteristics of that tier. This further motivates us towards a hierarchical data compression engine that can boost the overall performance of scientific applications.

III. Motivation and Problem Statement

To motivate our approach of introducing data compression to a multi-tiered storage environment, we investigated Vector Particle-In-Cell (VPIC) [46], a general-purpose simulation code for modeling kinetic plasmas in spatial multi-dimensions. In this application, each process produces data of simulated particles and writes them, at the end of each time-step, to storage. To visualize the problems we identified, we run VPIC with the following configuration: 2560 processes execute VPIC in 16 timesteps, each process produces 1GB of data for an overall data size of 8 TB organized in an HDF5 file. We run this test on Ares cluster at the Illinois Institute of Technology [47]. As Ares cluster has a multi-tiered storage system, we use Hermes [10] library to perform hierarchical buffering. We configure Hermes to use 16 GB of main memory, 32 GB of NVMe, and 2 TB of Burst Buffers, and lastly a 24-node OrangeFS for PFS. Figure 1 shows the results with X-axis showing the different configurations tested, Y-axis time elapsed in seconds, and the secondary Y2-axis the achieved compression ratio when applied. As a baseline, we run VPIC on top of the PFS with no compression.

Results show that without multi-tiered buffering and compression, VPIC takes 4270 seconds to complete. When VPIC runs with multi-tiered buffering enabled (i.e., on top Hermes), the application's runtime is optimized by 2.5x over the baseline PFS. When compression is enabled, significant application performance variability can be observed. Specifically, when light compression is applied (`Brotli`), an 1.93x reduction in the execution time is achieved by the smaller data size (i.e., 2x compression ratio in only 90 sec compression time). On the other hand, when heavy compression is applied (`Zlib`), the benefit of compression

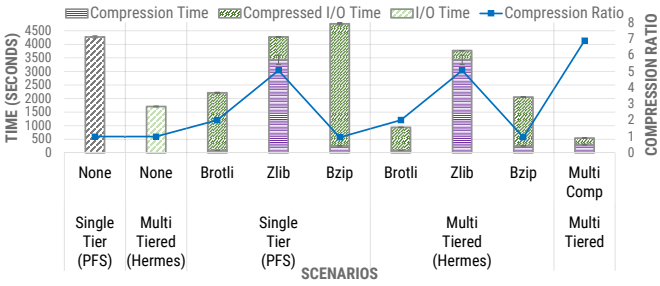


Fig. 1. VPIC running with data compression

is offset by the prolonged compression time (i.e., 5x compression ratio but in 3431 sec compression time). Finally, when unsuitable compression is applied, we observe that the data size might not decrease. In some cases, compressed data size might even be bigger than the uncompressed data [48]. For instance, VPIC data is not compressible by `Bzip`'s algorithm, hence, the longer I/O time observed. This performance variability can be caused by the ability of each compression library to apply meaningful compression [14].

It is clear that, both optimizations (i.e., multi-tiered buffering and data compression) should be combined for a greater overall benefit since, if compression is applied, more data can fit in the upper tiers of the storage hierarchy. For example, as it can be seen in the figure, VPIC runs roughly 2x faster, when both multi-tiered buffering and `Brotli` compression are enabled at the same time, compared to applying multi-tiered buffering or data compression individually. However, applying the same compression to all tiers of the hierarchy can lead to missed potential. Carefully choosing the appropriate compression algorithm for each tier can lead to the best performance. However, combining data compression and multi-tiered buffering is non-trivial. Matching compression metrics to the performance characteristics of each tier (i.e., a faster, but smaller in capacity, tier should perform more data compression than a slower, but larger in capacity, tier) is extremely challenging but rather rewarding, as it can be seen in the figure. This dynamic matching would both increase the resource utilization of the storage hierarchy and enable more effective data compression.

Problem Formulation: HCompress aims to improve the utilization of ITS by intelligently combining multi-tiered systems with data compression. Consider an application consisting of n I/O tasks with C compression libraries in a hierarchical environment with L storage tiers. We can formulate the optimization problem as shown in Table I. The objective function tries to minimize the overall time by compressing more data on higher tiers. Higher tiers have a smaller index. A global minima will occur when most of the data fits in higher tiers. This depends on the workload priorities as well as tier characteristics such as the tier's available capacity, access latency, and bandwidth. Furthermore, the objective function also considers the possibility of no compression since under certain system configurations, data compression might hurt the overall performance of the application. In the table, constraints 1-3 ensure a small number of sub-problems ensuring the low cost of the dynamic programming algorithm. Additionally,

TABLE I
PROBLEM FORMULATION

Given	i, an I/O task defined as "Applying Compression + Performing I/O"
	C, a set of compression algorithms with each element c
	t_c, compression time for algorithm c
	t_d, decompression time for algorithm c
	r_c, compression ratio for algorithm c
	L, a set of tiers with each element l
Define	P, indivisible sub-tasks of task i
	$Size(p)$, size of piece p
	$Concurrency(L)$, sum of hardware lanes in all tiers
	$Length(x)$, length of vector x
	$Duration(p, c, l)$, the time taken to execute p sub-task with c compression on l tier.
Minimize	$\sum_{p=0}^{P} Duration(p, c, l)$
Subject to (constraints)	$Size(p) \mod 4096 = 0$
	$Length(P) \leq Concurrency(T)$
	$Length(P) \leq Length(L)$
	$r_c \geq 1$
	$Size(p) \leq Size(l)$

sub-problems are highly reusable which further reduces the complexity of the algorithm. Constraint 4 ensures the selected compression will actually result in data reduction. Finally, constraint 5 guarantees that a sub-task can fit in a target tier.

IV. HCompress: Hierarchical Data Compression

HCompress is a data compression engine that leverages the existence of multiple tiers of the storage hierarchy to holistically optimize I/O operations and boost application performance. The main idea behind HCompress is to optimally match an I/O request with an appropriate compression algorithm and a suitable multi-tiered data placement. In other words, HCompress employs an intelligent, hierarchy-aware compression and data placement (HCDP) algorithm. To achieve this, HCompress uses an input data analyzer, a compression manager that has access to a corpus of compression libraries, a system monitor, a compression performance predictor, and an HCDP selection engine. Each I/O request is transformed into a task as a data buffer, operation tuple (e.g., compress and write or read and decompress). HCompress analyzes the task data to identify data attributes such as data type and format. HCompress also analyzes the system to identify the performance characteristics of each available tier. The selection engine then combines this knowledge to produce a compression and data placement schema which is finally passed to the compression manager for execution. HCompress is designed with the following principles in mind: 1) **Hierarchy-aware:** HCompress should apply the appropriate compression algorithm per tier by leveraging the different tier performance characteristics. This increases the effectiveness of compression by introducing different compression libraries on different tiers, thereby increasing the utilization of the hierarchy. 2) **Dynamic:** HCompress should be able to dynamically choose the required compression library with negligible cost and reconfigure itself for various requirements of the application transparently. 3) **Flexible:** HCompress should be able to unify all interfaces of the compression libraries by defining a common platform to interact with. This would allow the end-users to easily add new libraries to HCompress.

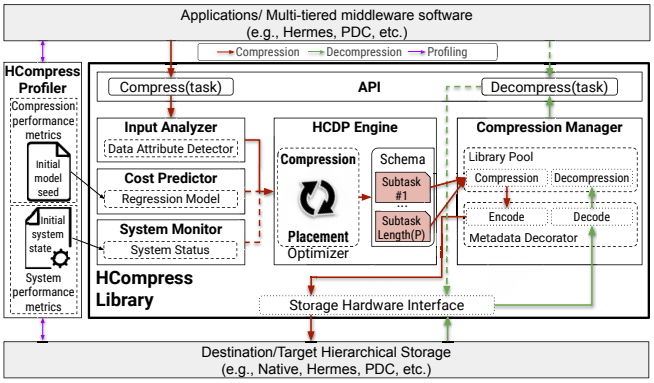


Fig. 2. HCompress design

A. Design and Architecture

Figure 2 shows the design of the HCompress ecosystem[1], the main library, and an external profiling tool. Applications can be compiled with the library (libhcompress.so) and use its native API (e.g., compress()/decompress()). To increase user-productivity and support legacy code, the HCompress library can also be dynamically linked using LD_PRELOAD. This way, HCompress can transparently intercept I/O calls and redirect them to its internal API. Our proposed design incorporates modern extreme scale system designs with compute node-local NVMe drives, a shared tier of burst buffers, and a remote PFS. The information about the tiers (e.g., bandwidth, device location, interface, etc.,) is provided by the user. However, the design of HCompress is generic and works with n-tiers of storage hierarchy. The flow of HCompress operations is as follows.

In the case of *compression* (red arrows in the figure), the application uses the HCompress library API and then the associated data is passed for analysis to the Input Analyzer (IA) which identifies the data attributes, namely data-type, distribution of data, and represented format. The Compression Cost Predictor (CCP) maintains a table of expected compression costs for each combination of the above data attributes. Furthermore, a System Monitor (SM) maintains the current system status, namely available tiers and their respective remaining capacity. The HCDP engine uses these attributes, the predicted compression cost, and the current system status to produce an optimal compression and placement schema. A schema consists of P sub-tasks, where each sub-task p contains the target tier along with the ideal compression library given by the optimization. The Compression Manager (CM) executes the schema by applying the indicated compression for each sub-task and encodes the compressed data with the appropriate metadata which is later used for decompression. Finally, the Storage Hardware Interface (SHI) is invoked to perform I/O on the multi-tiered storage. In the case of *decompression* (green arrows in the figure), the API will invoke the SHI to read the compressed data from the storage source. The CM will decode the metadata to identify which compression library was applied. It then decompresses the data and delivers it back to the application.

[1]https://bitbucket.org/scs-io/hcompress

Furthermore, HCompress utilizes an external profiling tool, HCompress Profiler (HP). The major objective of HP is to provide assistance on improving the accuracy of the predictive models used by the main library. It provides an initial seed, in the form of a JSON file, that all models use to bootstrap. This seed acts as a knowledge repository for HCompress to choose the correct compression for a given input. HP runs before the application starts to generate the seed by evaluating the performance of each compression library with a variety of input data (predefined or user-provided). Additionally, HP performs a discovery and benchmarking of the available hardware to produce a system signature which reflects the performance characteristics of the storage hierarchy. Lastly, in addition to the optimization constraints mentioned in section III, HCompress builds a light-weight dynamic mechanism to switch the compression libraries at run-time. Our prototype implementation of HCompress builds on top of our previous work, Hermes [10] and Ares [14].

B. Application Programming Interface

HCompress's Application Programming Interface (API) has two responsibilities: a) define the hierarchical compression interface, and b) to transparently intercept application I/O calls. To achieve the former, HCompress defines a simple `Compress(task)` and `Decompress(task)` API which performs the compression and decompression operations respectively. For the latter, all I/O calls such as POSIX and HDF5 (e.g., fopen, H5Dwrite, etc.,) are intercepted using dynamic linking, which internally calls the native API. Additionally, the API is responsible to intercept application's initialization (MPI_Init()) and finalization (MPI_Finalize()) calls which allows the main library to initialize its components (e.g., metadata, engine repository, establish connections to the external I/O clients, load compressors, etc.). On application finalization, HCompress frees all in-memory structures and stores the updated JSON back to storage for future reference.

C. Input Analyzer

The Input Analyzer (IA) is responsible for deducing the input data characteristics such as type, distribution, and format. For data-type and format inference, HCompress uses state-of-the-art techniques such as sub-sampling, binary decoding, and introspection [14]. The IA also examines the content distribution (as certain distributions are more compressible [49]) of the data and classifies each input buffer as Normal, Gamma, Exponential or Uniform. Distribution detection is performed statically using techniques such as sub-sampling and random partitioning [50]. Lastly, several data attributes can be easily obtained using metadata parsing of self-described portable data representations (e.g., HDF5, NetCDF, Avro, RDD, Parquet, etc.,) used in most scientific and cloud applications. Hence, in most practical cases, the IA is extremely fast and accurate.

D. Compression Cost Predictor

The Compression Cost Predictor (CCP) is mainly responsible to provide the HDCP engine with an Expected Compression Cost (ECC) for a given input. The ECC is defined as a 3-tuple of compression speed (in MB/s), decompression speed, and compression ratio (original over compressed size). The CCP employs a linear regression model, implemented using the `dlib` library, which, given a set of inputs, calculates the expected cost for each compression library. The model is initialized with the seed, generated by the HCompress Profiler, and evolves through time. The model input is: data-type (e.g., integer), data-format (e.g., csv), compression library (e.g., bzip2), and distribution (e.g., gamma). The model output is the ECC. Based on the predefined initial seed, our model demonstrated a good fit with an adjusted R^2 value of 94%, all variable p values less than 0.02, and a high f-statistic score of 928.13. However, on real datasets, we observed that the R^2 value drops to 83%. This is due to the fact that different datasets might have a different data distribution even when they have the same data type which, in turn, may result in different compression cost. During our implementation, we observed that, when the data distribution changes from uniform to gamma, the accuracy of our model may vary by $\pm 11\%$. To stabilize its accuracy, we incorporated a feedback mechanism using reinforcement learning. For every n operations (n is configurable), the compressors communicate the actual compression cost back to the CCP (i.e., feedback loop) which updates its model with the new observations. This mechanism increases the R^2 value to an average of 96% in all scenarios tested. This makes the model learn and grow as the application runs. At the end of application, we store the latest model back to the JSON seed so that it can be reused in future runs.

E. System Monitor

The System Monitor (SM) is responsible for timely providing the status of the storage hierarchy. The status of the hierarchy includes availability (boolean), load (queue size) and remaining capacity (in bytes) per tier. An initial system state is provided by the HCompress Profiler. Henceforth, the SM constantly monitors the changes in status of each tier by periodically measuring the above metrics. The SM is implemented using a background thread that executes a collection of system tools (i.e., `du` and `iostat` found in most Linux distributions). The status of the hierarchy is required by the HDCP engine to optimally derive the placement of each new incoming I/O task.

F. Hierarchical Compression and Data Placement Engine

The HCDP Engine (engine from now on) is the brain of HCompress. This component is responsible for devising the compression and placement schema for an incoming I/O task. The engine receives the input data characteristics from the IA, the expected compression cost from the CCP, and the current system status from the SM. It runs a recursive multi-dimensional optimizer (described below) to produce one or more sub-tasks for each input task. The set of created sub-tasks comprises of a schema that is passed to the Compression Manager (CM) for execution. Furthermore, the engine can dynamically tune its mappings to adaptively match the application's compression requirements.

1) Hierarchical Compression and Placement Algorithm (HCDP): When a task is received, the engine recursively matches and places data for all combinations of target tier

and compression library. During the calculation of the cost of each sub-problem, the engine pulls the current status of the tier and estimated compression cost from the SM and CCP respectively. For every combination, if the compressed data can fit in an upper layer, then it will be added to the optimization space as a sub-problem. Otherwise, the task will be split in two parts in multiples of 4096 bytes: one that can fit in the remaining capacity of the current tier and one holding the rest of the I/O task. This satisfies constraints $1, 2$, and 3 of the problem statement described in Section III. Our choice of 4096 bytes is motivated from the page-size of RAM and the block size of modern storage devices such as NVMe SSDs. This will lead to aligned I/O. More importantly, however, this choice makes the memoization highly effective as the sub-problems would be reusable. One critical point to note is that "no-compression" is one of the choices available to the engine. This is because in certain circumstances, performing compression might hurt the overall performance [48]. The solution can be expressed as a recursive dynamic programming optimization. The mathematical formulation is given in equations 1 and 2. Additionally, the cost function of this optimization is given by equations 3 and 4.

$$Match(i,l,c) = \begin{cases} Place(i,l,c) & s_l \geq s_{ic} \\ Place(i_l,l,c) + Match(a_{ic},l+1,c) & otherwise \end{cases} \quad (1)$$

$$Place(i,l,c) = Min \left\{ \begin{matrix} t(i,l), \\ t(i,l,c), \\ Match(i,l+1,c), \\ Match(i,l,c+1) \end{matrix} \right\} \quad (2)$$

$$t(i,l) = \frac{s_i}{b_l} \quad (3)$$

$$t(i,l,c) = w_c * t_c + t(i,l) - w_r * \frac{t(i,l) * (r_c - 1)}{r_c} + w_d * t_d \quad (4)$$

$Match$ represents the matching of I/O task i to a tier l with a compression library c. l is the index of a tier in the set of all tiers L, where lower values of l represent upper tiers (e.g., $l = 0$ represents RAM). c represents the index of the compression library in the set of all compression libraries C, with $c = 0$ representing no compression. Also, w_c, w_d, and w_r represent weights for t_c, t_d and r_c respectively. $Place$ represents the placement of task i on tier l with compression library c. $Place$ assumes that the task can fit in the current tier after applying compression c. s_{ic} is the size of task i after applying compression c given by $s_{ic} = \frac{s_i}{r_c}$. a_{ic} represents the remaining size of task i given by $a_{ic} = s_{ic} - s_l$. Finally, $t(i,l,c)$ and $t(i,l)$ are the cost functions of the time to execute task i on tier l with and without compression respectively. The above dynamic programming optimization is almost constant with time complexity of $O(2 * (len(L))^2)$, where $len(L)$ is very small (in the order of tens). Hence, the time complexity of HDCP algorithm is practically $O(1)$.

2) Application compression priorities: The engine chooses the optimal compression-to-tier mapping for a given I/O task based on the equations 3 and 4. When compression is applied, the cost function is dependent on three factors: compression time, decompression time, and reduced I/O time. Additionally, the cost function has three weighted components attached to the above metrics. These weights, which are configurable at runtime, dictate the priority of the application. Examples of such weight values are shown in Table II. Users can define a global weighting scheme in the JSON seed. More advanced users can leverage HCompress API to dynamically change these weights at runtime.

TABLE II
COMPRESSION PRIORITIZATION EXAMPLES.

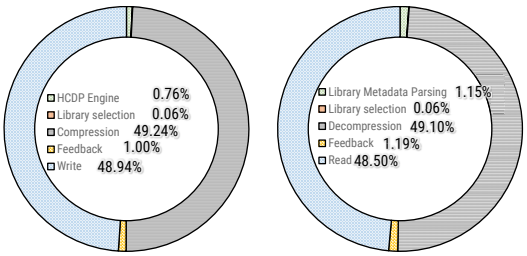
Scenario	Ratio	Compression Speed	Decompression Speed
Asynchronous I/O	0	1	0
Archival I/O	1	0	0
Read After Write	0.4	0.3	0.3

G. Compression Manager

The Compression Manager (CM) is responsible for managing the interface to several compression libraries. It unifies several compression APIs under a common interface and decorates compressed data with enriched metadata.

1) Compression library unification: In HCompress, the unification of the compression libraries is performed by the Compression Library Pool (CLP). The CLP has three main parts: 1) Compression Library Interface: defines a set of virtual functions which encapsulates the functionality of a compression library along with a universal signature, 2) Compression Library Implementation: implements the interface we defined for a specific compression library, and finally, 3) Compression Library Factory: chooses a specific implementation based on a given input. The Compression Library Interface uses a template pattern to define a generic interface which can be easily and efficiently specialized. It consists of the following compression libraries [14]: bzip2, zlib, huffman, brotli, bsc, lzma, lz4, lzo, pithy, snappy, and quicklz. It essentially implements the higher-level functions defined by the interface into compression-library-specific calls along with error handling as defined by the interface. Finally, the Compression Library Factory uses an adapter-plus-factory design pattern to efficiently switch between libraries. The choice of the compression library is dependent on the constant provided by the caller class. The factory is the only place from which caller can instantiate the implementations. This enables the CLP to easily call new libraries dynamically without changing existing code of the caller in $O(1)$ complexity.

2) HCDP Algorithm metadata: Since the HDCP engine can choose different compression libraries for different inputs of data and for different tiers, HCompress needs to maintain which compression library was applied, how it was applied (i.e., offsets of the input buffers), and the original size of the uncompressed data. This is accomplished by utilizing a small header (i.e., 16-bytes) attached to each sub-task which holds this info as a 4-tuple of {*start-offset, length, compression library, resulting size*}. This allows the Compression Manager to easily determine which compression library to use to later decompress the data. This method is efficient and highly scalable as each application process can independently identify the compression library from the data itself.



(a) Write Operation (b) Read Operation

Fig. 3. HCompress anatomy of operations.

V. Evaluation

A. Methodology and Experimental Setup

1) Configurations: We ran all of our experiments on the Ares supercomputer at the Illinois Institute of Technology. The entire cluster runs on a 40 GBit Ethernet with RoCE capabilities. We configure the buffers, unless specified otherwise, to fit 20% of data in local RAM, 30% in local NVMe and rest in burst buffers. Cluster specifications are shown in Table III and Table IV shows the configurations tested.

TABLE III
Testbed Specifications.

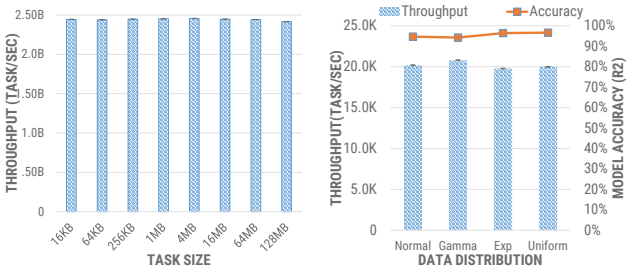
Node Type	CPU	RAM	Disk
Compute x64	Intel Xeon Silver 4114 @ 2.20GHz	DDR4 96GB	512GB NVMe SSD
Burst Buffers x4	AMD Dual Opteron 2384 @ 2.7Ghz	DDR3 64GB	2x512GB SSD
Storage x24	AMD Dual Opteron 2384 @ 2.7Ghz	DDR3 32GB	2TB HDD

TABLE IV
Test Configurations.

Test case	Abbreviation	Hierarchical	Compression
Baseline vanilla PFS	BASE	No	No
Single-tier with compression	STWC	No	Yes
Multi-tiered without compression	MTNC	Yes	No
HCompress	HC	Yes	Yes

2) Workloads: To evaluate HCompress, we first use micro-benchmark workloads to measure the performance of internal components. The micro-benchmarks are from HDF5 source code [51], where each process creates a shared HDF5 file and reads/writes an independent, but overall contiguous, block of data. Additionally, we use these workloads to evaluate the impact of data compression on multi-tiered storage and vice-versa. Finally, we use I/O workloads from real scientific applications to compare HCompress against Hermes [10], a state-of-the-art multi-tiered storage software solution. The scientific workloads include VPIC-IO [46] and BD-CATS-IO [52], which 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 $n * 8 * 2^{20} * 32$, where n is the number of MPI processes. BD-CATS-IO reads properties from datasets similar to those produced by VPIC for a parallel clustering algorithm to identify the different groups of particles. In all cases, we set the workload priority to equal for compression metrics, unless specified otherwise.

3) Performance metrics: We measured the time required to open, write, read, and close a file. Additionally, we define throughput as the rate of requests processed per second. Finally, all tests were executed five times and we report the average along with standard deviation.



(a) HCDP Algorithm (b) Compression Cost Predictor

Fig. 4. HCompress internal components performance.

B. HCompress Internal Component Evaluation

In this section, we first evaluate the anatomy of HCompress write and read I/O tasks to better understand any associated overheads of our approach. We then evaluate the performance of internal components. Finally, we evaluate the impact of data compression on multi-tiered storage and vice-versa.

1) HCompress Overhead Analysis: Each I/O task in HCompress solves the problem of where to place data in the hierarchy and which compression algorithm to use to compress the data. In this test, we perform 1K tasks of 1MB and present the overall time breakdown into its components in Figure 3 (write 3(a) and read 3(b)). We observe that, 98% of the time, both read and write operations are spent in I/O and compression/decompression. A fraction of the overall time is spent by the HCDP engine, library selection, and feedback combined. This shows that the cost of the mapping and library selection is low and remained on average below 2%. Specifically, the mapping engine is 0.76%, the library selection takes 0.06%, and feedback takes 1% of the overall time. A similar trend is shown in the read operation, where library metadata parsing only takes 1.15%, the library selection takes 0.06%, and feedback takes 1.19% of the overall time.

2) Performance of the HCDP Engine: The performance of the algorithm that the engine runs is very critical for the write path within HCompress. Hence, the algorithm should demonstrate high throughput of mapping various I/O tasks to their corresponding tiers and compression libraries. To evaluate this, we perform 8K writes of various task sizes and calculate the throughput of the engine. The throughput of the algorithm is shown in Figure 4(a). In this figure, the x-axis represents various task sizes and the y-axis shows the overall throughput (tasks/second). We can observe that until 4MB task size, the throughput of HCDP algorithm is almost constant at about 2.44 billion tasks per second. As the I/O size increases, the throughput drops by 2-3%. This is due to the fact that for bigger tasks, the HCDP algorithm has to split the task into several pieces (so that it can fit it into limited capacity tiers) which make it span across multiple tiers. Overall, this evaluation highlights that the HCDP algorithm is very light-weight and has high, constant throughput.

3) Performance of the Compression Cost Predictor: The CCP's model accuracy is critical for the optimality of the engine. As discussed in the section IV-D, we use a linear regression model with reinforcement learning to predict and learn the variables of the cost function. To evaluate the performance of

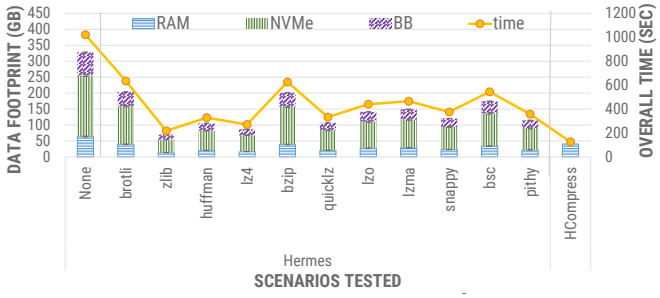


Fig. 5. Impact of Data Compression on Multi-tiered Storage

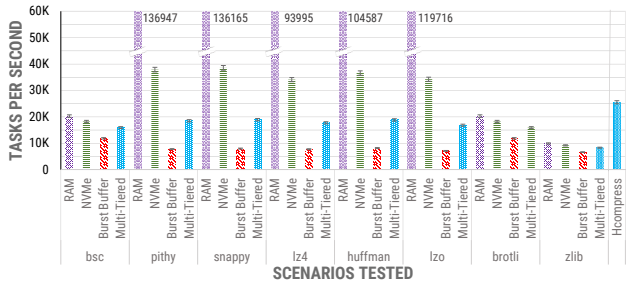


Fig. 6. Impact of Multi-tiered Storage on Data Compression

this component, we perform 8K write tasks of 1MB size using four different data distributions: uniform, normal, exponential, and gamma. Since different distributions will result in different compression ratios, compression times, and decompression times, this test will show the ability of the feedback mechanisms to learn new patterns and predict the cost accurately. In figure 4(b), the x-axis shows the various data distributions, the y-axis represents the throughput of the feedback engine, and the y2-axis shows the accuracy achieved. We can observe that, for all distributions, the model accurately predicts the compression cost (i.e., performance metrics). Specifically, it achieves an accuracy of 95.54%. Additionally, the feedback mechanism achieves a constant throughput at around 20K events per second. This is due to the fact that linear regression is not computationally intensive and can produce predictions quickly.

4) Impact of Data Compression on Multi-tiered Storage: To quantify the benefits of enabling adaptive data compression in multi-tiered storage, we run the following workload: 2560 MPI ranks in 64 nodes issue 128 write tasks, of 1MB per process, for a total size of 320GB. The hardware hierarchy is configured as follows: 64GB RAM, 192GB NVMe, and 2TB BB. Figure 5 shows the results. In x-axis, we compare HCompress against our baseline of Hermes's data placement without compression (depicted as *None*). Further, we compare against Hermes's data placement with compression enabled using several different libraries. Note that, in this test configuration, Hermes will first solve the data placement based on the uncompressed size of incoming I/O and then apply the selected compression. The y-axis shows the compressed data footprint in GB, depicted per tier, and the y2-axis shows the elapsed time in seconds. As it can be seen, applying compression reduces the amount of data written in each tier, decreasing the overall execution time. For instance, Hermes with `brottli` compressed the data down to 203GB and wrote it in 634 seconds. On the other hand, Hermes with `zlib` reduced even more the data footprint (i.e., 70GB) thus achieving the overall execution time of 218 seconds. The key observation we make for Hermes is that the tiers of the hierarchy will be under-utilized (e.g., Hermes with `lz4` only uses 17GB out of the 64GB available in RAM). This is due to the fact that Hermes solves the data placement *before* it applies compression. Contrary to this, HCompress places the data in the tiers of the hierarchy considering their compressed footprint which leads to significant performance gains by up to 8x compared to no compression and at least by 1.72x compared to other compression libraries.

5) Impact of Multi-tiered Storage on Data Compression: The benefits of data compression rise from balancing compression time with the reduction of I/O. Thus, this trade-off is very critical to the overall task duration. To better understand the impact of multiple tiers on the compression/decompression rate, we run the following workload: 2560 MPI ranks in 64 nodes, each rank issues 512 tasks, each task consists of compressing and writing 512KB and reading and decompressing it back. The total amount of data written and read is 600GB. For a single tier, the whole dataset can fit in each tier respectively. For multi-tier storage, the hierarchy is configured as follows: 32GB RAM, 96GB NVMe, and 1TB BB. Figure 6 shows the results. In x-axis, we compare HCompress against a collection of compression libraries applied per single-tier and for all available tiers (depicted as *multi-tiered*). The y-axis shows the throughput expressed in tasks per second. As it can be seen, there is significant throughput variability across tiers and across libraries. For instance, `bsc`, `brottli`, and `zlib` demonstrate a stable rate across tiers since their task completion time is determined by the compression and decompression time (i.e., they apply heavy compression). On the other hand, the throughput of `pithy`, `snappy`, `lz4`, `huffman`, and `lzo` is very sensitive to the change of tier since their task completion time is dominated by the I/O time (i.e., different tiers have different bandwidth resulting to different I/O rate). Once we move to a multi-tiered scenario, we can see that the achieved throughput gets balanced since the rate variability across libraries and tiers is averaged out. However, since a single compression library is applied on all tiers of the hierarchy, we identify the missed potential for further performance by mapping different libraries to different tiers. HCompress capitalizes on this observation and uses the best library for each tier leading to a higher throughput. Specifically, in this test, HCompress uses `pithy` on RAM, `snappy` on NVMe, and `brottli` on the burst buffers since these libraries achieved the best rate for the respective tier. In summary, HCompress achieved 1.4-3x higher throughput compared to other compression libraries applied on multi-tiered storage.

C. HCompress with Scientific Applications

1) VPIC-IO: Scientific simulations, such as VPIC [46], 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 how an application will benefit by HCompress. In VPIC-IO, each process writes eight variables with a total size of

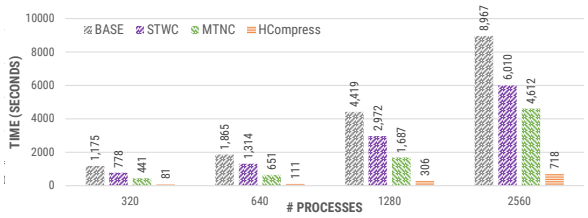


Fig. 7. VPIC-IO

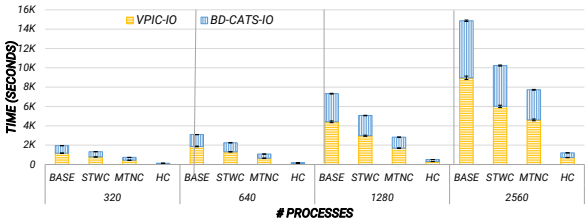


Fig. 8. VPIC-IO plus BD-CATS-IO workflow with 10 timesteps.

256MB in each time step. We run VPIC for 10 time steps. The hierarchical hardware is configured with 12.5 GB RAM and 25GB NVMe space which is insufficient for buffering data for more than 5 time steps. Thus, the remaining data has to be spilled to the next storage layer (e.g., burst buffers). To emulate the computation behavior, we manually add a CPU-intensive kernel of simple random matrix multiplications at an interval of 60 seconds between checkpoints. Additionally, as VPIC-IO is a write-only workload, we configure HCompress to prioritize compression time and compression ratio. The I/O time for our baseline (labeled as *BASE*) represents only the time required to write to the PFS for all the time steps. We compare HCompress under the configurations shown in Table IV. This test demonstrates the effect of applying an I/O optimization independently (i.e., enabling compression OR enabling multi-tiered buffering) and in combination (i.e., HCompress).

Figure 7 shows the results. We scale the number of processes from 320 processes to 2560 processes to test the scalability of each solution. The accumulated data of this test does not fit entirely in the DRAM and NVMe, and therefore will have to spill more than 60% of the data to the burst buffers. As it can be seen, applying compression before we write data to the PFS boosts the performance by 1.5x for the largest scale since the data footprint is reduced significantly before VPIC performs the write. Similarly, when enabling multi-tiered buffering (Hermes but without compression), the execution time reduced by 2x. Both optimizations work as expected, but they work in isolation. In contrast, HCompress compresses and places data in the hierarchy in an optimal way and is able to fit more data in the upper tiers, which boosts the performance even more. Results show a 12x improvement over the baseline and 7x on average over the other optimizations.

2) BD-CATS analytics: Scientific applications often involve workflows where data producers and consumers share data. To evaluate HCompress's support for scientific workflows, we use BD-CATS-IO to read data produced by VPIC-IO. We configure both programs to run with 10 time steps. Similar to the experiment in V-C, data produced in 10 time steps does not fit entirely in the DRAM + NVMe layers and has to spill over to the burst buffers. We evaluate HCompress's acceleration of I/O by comparing it with compression on single tier (STWC) and multi-tiered I/O buffering without compression (MTNC). Our baseline remains the vanilla PFS without any optimization applied. The execution is sequenced by BD-CATS-IO starting after VPIC-IO finishes. Additionally, as this workflow involves both write and read, we configure HCompress to prioritize all the three compression metrics equally. We show in Figure 8 the results of this test. As it can be seen, introducing single tier compression/decompression in the workflow boosts performance by approximately 1.5x over the baseline. Similarly, enabling multi-tiered buffering increases performance by 2.5x on average compared to the baseline since it utilizes upper (faster) tiers to buffer the data. On the other hand, HCompress is 7x faster than both STWC and MTNC due to its optimal data reduction and placement. The benefit of combining both compression and hierarchical buffering is compounding and significantly outperforms each optimization individually. In summary, workflows that consist of read after write patterns like VPIC and BD-CATS are expected to benefit greatly from HCompress.

VI. Conclusion

Modern scientific applications operate on massive amounts of data. The involved I/O is often the bottleneck in these applications. To alleviate this problem, data access optimizations have been proposed, namely multi-tiered I/O buffering and data compression. These two optimizations assume the presence of intermediate temporary scratch space and their effectiveness highly depends on its space availability. To increase the space in ITS, multi-tiered storage has added extra tiers to the storage hierarchy while compression aims to minimize the data footprint. However, these approaches are orthogonal and do not co-exist, which comes at a detriment to both. In this paper, we present HCompress, a hierarchical data compression engine for multi-tiered storage environments. HCompress optimally combines the above optimizations under the same runtime. We introduce a dynamic hierarchical data compression and placement algorithm which maps the spectrum of compression libraries to the spectrum of tiers of the hierarchy. Our performance evaluation demonstrates the impact of multiple tiers on the data compression, the impact of applying compression to the hierarchical storage, and the impact of doing both. HCompress can improve I/O performance significantly for scientific applications and workflows as shown in this study. Specifically, HCompress performs 12x faster over a baseline of vanilla I/O on top of a PFS, and up to 7x faster than other competing solutions. As a future step, we plan to test our system on larger-scale supercomputers with more applications. We also plan to incorporate this technology into the Hermes ecosystem and offer it as a readily available solution to users.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant no. OCI-1835764 and CSR-1814872.

REFERENCES

[1] A. Katal, M. Wazid, and R. Goudar, "Big data: issues, challenges, tools and good practices," in *2013 Sixth international conference on contemporary computing (IC3)*. IEEE, 2013, pp. 404–409.

[2] R. Kitchin, "Big Data, new epistemologies and paradigm shifts," *Big data & society*, vol. 1, no. 1, p. 2053951714528481, 2014.

[3] A. Kumar, M. Boehm, and J. Yang, "Data management in machine learning: Challenges, techniques, and systems," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1717–1722.

[4] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[5] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in hpc storage systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2012, p. 7.

[6] Y. J. Stephanedes and A. P. Chassiakos, "Application of filtering techniques for incident detection," *Journal of transportation engineering*, vol. 119, no. 1, pp. 13–26, 1993.

[7] L. Petersen, P. Minkkinen, and K. H. Esbensen, "Representative sampling for reliable data analysis: theory of sampling," *Chemometrics and intelligent laboratory systems*, vol. 77, no. 1-2, pp. 261–277, 2005.

[8] H. K. Reghbati, "Special feature an overview of data compression techniques," *Computer*, no. 4, pp. 71–75, 1981.

[9] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager *et al.*, "Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1033–1042.

[10] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. USA: ACM, 2018, pp. 219–230.

[11] T. Wang, S. Byna, B. Dong, and H. Tang, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC," in *IEEE International Conference on Cluster Computing*. IEEE, 2018, pp. 134–144.

[12] D. Salomon, *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[13] K. Sayood, *Introduction to data compression*. Morgan Kaufmann, 2017.

[14] H. Devarajan, A. Kougkas, and X.-H. Sun, "An Intelligent, Adaptive, and Flexible Data Compression Framework," in *Proceedings of the IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid'19)*. Larnaca, Cyprus: IEEE, 2019.

[15] P. Valduriez, M. Mattoso, R. Akbarinia, H. Borges, J. Camata, A. Coutinho, D. Gaspar, N. Lemus, J. Liu, H. Lustosa *et al.*, "Scientific Data Analysis Using Data-Intensive Scalable Computing: the SciDISC Project," in *LADaS: Latin America Data Science Workshop*, no. 2170. CEUR-WS. org, 2018.

[16] Y. Ma, H. Wu, L. Wang, B. Huang, R. Ranjan, A. Zomaya, and W. Jie, "Remote sensing big data computing: Challenges and opportunities," *Future Generation Computer Systems*, vol. 51, pp. 47–60, 2015.

[17] J. C. Brustoloni and P. Steenkiste, "Effects of Buffering Semantics on I/O Performance," in *In Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1996, pp. 277–291.

[18] 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 *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 3.

[19] S. Dar, M. J. Franklin, M. Jonsson *et al.*, "Semantic data caching and replacement," in *VLDB*, vol. 96, 1996, pp. 330–341.

[20] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of IEEE conference on Supercomputing*. IEEE Press, 2008, p. 44.

[21] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, no. 3, pp. 277–290, 2010.

[22] F. Pellizzer, A. Pirovano, F. Ottogalli, M. Magistretti, M. Scaravaggi, P. Zuliani, M. Tosi, A. Benvenuti, P. Besana, S. Cadeo *et al.*, "Novel/spl mu/trench phase-change memory cell for embedded and stand-alone non-volatile memory applications," in *Digest of Technical Papers. 2004 Symposium on VLSI Technology, 2004.* IEEE, 2004, pp. 18–19.

[23] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[24] I. Baek, M. Lee, S. Seo, M. Lee, D. Seo, D.-S. Suh, J. Park, S. Park, H. Kim, I. Yoo *et al.*, "Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses," in *IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004.* IEEE, 2004, pp. 587–590.

[25] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *ACM SIGARCH computer architecture news*, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 453–464.

[26] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An interference-aware dynamic i/o scheduler for shared non-volatile burst buffers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 290–301.

[27] NERSC, "Technical Specifications," 2017. [Online]. Available: https://www.nersc.gov/users/computational-systems/cori/configuration/

[28] Cray, "Datawarp documentation," 2016. [Online]. Available: https://pubs.cray.com/browse/datawarp/software

[29] J. Lujan, "Technical Specifications," 2015. [Online]. Available: https://www.lanl.gov/projects/trinity/specifications.php

[30] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *012 ieee 28th symposium on mass storage systems and technologies (msst)*. IEEE, 2012, pp. 1–11.

[31] F. Schmuck and R. Haskin, "A shared-disk file system for large computing clusters. Proceedings of the 1st USENIX Conference on File and Storage Technologies," *USENIX Association*, p. 19, 2002.

[32] G. K. Lockwood, D. Hazen, Q. Koziol, R. Canon, K. Antypas, J. Balewski, N. Balthaser, W. Bhimji, J. Botts, J. Broughton *et al.*, "Storage 2020: A vision for the future of hpc storage," 2017.

[33] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, and et Al, "Toward Scalable and Asynchronous Object-centric Data Management for HPC," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '18. IEEE, 2018, pp. 113–122. [Online]. Available: https://doi.org/10.1109/CCGRID.2018.00026

[34] A. Ovsyannikov, M. Romanus, B. Van Straalen, G. H. Weber, and D. Trebotich, "Scientific workflows at datawarp-speed: accelerated data-intensive science using NERSC's burst buffer," in *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, 2016, pp. 1–6.

[35] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick *et al.*, "A case for core-assisted bottleneck acceleration in gpus: enabling flexible data compression with assist warps," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 41–53, 2016.

[36] H. Yeo and C. H. Crawford, "Big data: Cloud computing in genomics applications," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 2904–2906.

[37] F. Pan, Y. Yue, J. Xiong, and D. Hao, "I/o characterization of big data workloads in data centers," in *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, 2014, pp. 85–97.

[38] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms, "Scalable parallel i/o on a blue gene/q supercomputer using compression, topology-aware data aggregation, and subfiling," in *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014, pp. 107–111.

[39] D. Zhao, J. Yin, K. Qiao, and I. Raicu, "Virtual chunks: On supporting random accesses to scientific data in compressible storage systems," in *2014 IEEE International Conference on Big Data*, 2014, pp. 231–240.

[40] A. H. Baker, H. Xu, J. M. Dennis, M. N. Levy, D. Nychka, S. A. Mickelson, J. Edwards, M. Vertenstein, and A. Wegener, "A Methodology for Evaluating the Impact of Data Compression on Climate Simulation Data," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 203–214. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600217

[41] D. Laney, S. Langer, C. Weber, P. Lindstrom, and A. Wegener, "Assessing the effects of data compression in simulations using physically motivated metrics," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[42] K. Masui, M. Amiri, L. Connor, M. Deng, M. Fandino, C. Hfer, M. Halpern *et al.*, "A compression scheme for radio data in high performance computing," *Astronomy and Computing*, vol. 12, pp. 181 – 190, 2015.

[43] Google, "Snappy — a fast compressor/decompressor," 2019. [Online]. Available: http://google.github.io/snappy/

[44] Austin Seipp, "Fast compression library for C, C and Java," 2019. [Online]. Available: http://www.quicklz.com/

[45] M. McDaniel and M. H. Heydari, "Content based file type detection algorithms," in *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. IEEE, 2003, pp. 10–pp.

[46] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter *et al.*, "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[47] Scalable Computing Lab, Illinois Tech, "Ares Supercomputer @ IIT," 2019. [Online]. Available: http://www.cs.iit.edu/ scs/resources.html

[48] M. A. Roth and S. J. Van Horn, "Database compression," *ACM Sigmod Record*, vol. 22, no. 3, pp. 31–39, 1993.

[49] R. Gribonval, V. Cevher, and M. E. Davies, "Compressible distributions for high-dimensional statistics," *IEEE Transactions on Information Theory*, vol. 58, no. 8, pp. 5016–5034, 2012.

[50] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 65:1–65:35, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2896499

[51] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 2011, pp. 36–47.

[52] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, P. Dubey *et al.*, "BD-CATS: big data clustering at trillion particle scale," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.