Anthony Kougkas *et al.* I/O Acceleration via Multi-Tiered Data Buffering and Prefetching JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY ?????: 1–33 ??????. DOI ???????

I/O Acceleration via Multi-Tiered Data Buffering and Prefetching

Anthony Kougkas[1,*], Hariharan Devarajan[1], and Xian-He Sun[1,**]

[1] *Illinois Institute of Technology, Department of Computer Science, Chicago 60616, USA*

E-mail: akougkas@iit.edu, hdevarajan@hawk.iit.edu, sun@iit.edu

Received June 21, 2019; revised August 25, 2019.

Abstract Modern High-Performance Computing (HPC) systems are adding extra layers to the memory and storage hierarchy, named deep memory and storage hierarchy (DMSH), to increase I/O performance. New hardware technologies, such as NVMe and SSD, have been introduced in burst buffer installations to reduce the pressure for external storage and boost the burstiness of modern I/O systems. The DMSH has demonstrated its strength and potential in practice. However, each layer of DMSH is an independent heterogeneous system and data movement among more layers is significantly more complex even without considering heterogeneity. How to efficiently utilize the DMSH is a subject of research facing the HPC community. Further, accessing data with a high-throughput and low-latency is more imperative than ever. Data prefetching is a well-known technique for hiding read latency by requesting data before it is needed to move it from a high-latency medium (e.g., disk) to a low-latency one (e.g., main memory). However, existing solutions do not consider the new deep memory and storage hierarchy and also suffer from under-utilization of prefetching resources and unnecessary evictions. Additionally, existing approaches implement a client-pull model where understanding the application's I/O behavior drives prefetching decisions. Moving towards exascale, where machines run multiple applications concurrently by accessing files in a workflow, a more data-centric approach resolves challenges such as cache pollution and redundancy. In this paper, we present the design and implementation of *Hermes*: a new, heterogeneous-aware, multi-tiered, dynamic, and distributed I/O buffering system. Hermes enables, manages, supervises, and, in some sense, extends I/O buffering to fully integrate into the DMSH. We introduce three novel data placement policies to efficiently utilize all layers and we present three novel techniques to perform memory, metadata, and communication management in hierarchical buffering systems. Additionally, we demonstrate the benefits of a truly hierarchical data prefetcher that adopts a server-push approach to data prefetching. Our evaluation shows that, in addition to automatic data movement through the hierarchy, Hermes can significantly accelerate I/O and outperforms by more than **2x** state-of-the-art buffering platforms. Lastly, results show **10-35%** performance gains over existing prefetchers and over **50%** when compared to systems with no prefetching.

Keywords I/O buffering, Heterogeneous buffering, Layered buffering, Deep memory hierarchy, Burst buffers, Hierarchical data prefetching, Data-centric architecture

1 Introduction

Data-driven science is a reality and in fact, is now driving scientific discovery [1]. An International Data Corp. (IDC) report [2] predicts that by 2025, the global data volume will grow to 163 zettabytes (ZB), ten times the 16.1ZB of data generated in 2016. The evolution of modern storage technologies is driven by the increasing ability of powerful High-Performance Computing (HPC) systems to run data-intensive problems at larger scale and resolution. In addition, larger scientific instruments and sensor networks collect extreme amounts of data and push for more capable storage systems [3]. Modern I/O systems have been developed and highly optimized through the years. Popular interfaces and standards such as POSIX I/O, MPI-IO [4], and HDF5 [5] expose data to the applications and al-

Regular Paper
Funded by the National Science Foundation under Grants No. OCI-1835764 and CSR-1814872.
*Corresponding Author - ACM Member **IEEE Fellow

low users to interact with the underlying file system through extensive APIs. In a large scale environment, the underlying file system is usually a parallel file system (PFS) with Lustre [6], GPFS [7], PVFS2 [8] being some popular examples. However, as we move towards the exascale era, most of these storage systems face significant challenges in performance, scalability, complexity, and limited metadata services [9, 10], creating the so called *I/O bottleneck* which will lead to less scientific productivity [11, 12].

To reduce the I/O performance gap, modern storage subsystems are going through extensive changes, by adding additional levels of memory and storage in a hierarchy [13]. Newly emerging hardware technologies such as High-Bandwidth Memory (HBM), Non-Volatile RAM (NVRAM), Solid-State Drives (SSD), and dedicated buffering nodes (e.g., burst buffers) have been introduced to alleviate the performance gap between main memory and the remote disk-based PFS. Modern supercomputer designs employ such hardware technologies in a heterogeneous layered memory and storage hierarchy, we call *Deep Memory and Storage Hierarchy (DMSH)* [14, 15]. For example, Cori system at the National Energy Research Scientific Computing Center (NERSC)*, uses CRAY's Datawarp technology†. Los Alamos National Laboratory Trinity supercomputer‡ uses burst buffers with a 3.7 PB capacity and 3.3 TB/s bandwidth. Summit in Oak Ridge National Lab is also projected to employ fast local NVMe storage for buffering§.

As multiple layers of storage are added into HPC systems, the complexity of data movement among the layers increases significantly, making it harder to take advantage of the high-speed and low-latency storage systems [16]. Additionally, each layer of DMSH is an independent system that requires expertise to manage, and the lack of automated data movement between tiers is a significant burden currently left to the users [17]. Popular I/O middleware, such as HDF5, PnetCDF [18], and ADIOS [19], are configured to operating with the traditional memory-to-disk I/O endpoints. This middleware provides great value by isolating users from the complex effort to extract peak performance from the underlying storage system, but it will need to be updated to handle the transition to a multi-tiered I/O configuration [17]. Furthermore, optimizing read data access patterns is crucial to achieving computational efficiency. One popular practice employed is data prefetching. The effectiveness of data prefetching depends upon the ability to recognize data access patterns and to timely identify the data which should be prefetched. Therefore, both timeliness and accuracy are critical in the perceived performance of a data prefetcher. All prefetching solutions have to answer two main questions [20]: a) when to prefetch data, and b) what data to prefetch. Prefetching the wrong data or the right data at a wrong time not only does not help but actually hurts the overall performance [21]. Additionally, the presence of multiple tiers of the storage hierarchy raises a third question: *where to prefetch data*? There is a need to seamlessly and transparently support access to DMSH.

In this paper, we present the design and implementation of Hermes¶: a new, heterogeneous-aware, multi-tiered, dynamic, and distributed I/O buffering system. Hermes enables, manages, and supervises I/O buffering

*https://www.nersc.gov/users/computational-systems/cori/burst-bufer/

†http://www.cray.com/sites/default/files/resources/ CrayXC40-DataWarp.pdf

‡http://www.lanl.gov/projects/trinity/specifications.php

§https://tinyurl.com/y2676no5

¶Ancient Greek God of "*messaging and the transgression of boundaries*"

into DMSH and offers: a) *vertical and horizontal distributed buffering in DMSH* (i.e., access data to/from different levels locally and across remote nodes), b) *selective layered data placement* (i.e., buffer data partially or entirely in various levels of the hierarchy), c) *dynamic buffering via system profiling* (i.e., change the buffering schema dynamically by monitoring the system status such as capacity of buffers, messaging traffic, etc.). Hermes accelerates applications' I/O access by transparently buffering data in DMSH. Data can be moved through the hierarchy effortlessly and therefore, applications have a capable, scalable, and reliable middleware software to navigate the I/O challenges towards the exascale era. Lastly, by supporting both POSIX and HDF5 interfaces, Hermes offers ease-of-use to a wide-range of scientific applications. Hermes has been carefully designed to enable data-centric prefetching decision engine that utilizes system-generated events, while leveraging the presence of multiple tiers of storage, to perform hierarchical data placement at the required time. We build upon the observation that scientific workloads demonstrate a WORM data access model (i.e., write-once-read-many) [22], which is also true for BigData applications [23, 24, 25]. We also target modern scientific workflows that span across multiple applications in a pipeline of data processing. In such environments, data might be read multiple times across applications which might create severe issues for prefetching cache management. Cache pollution, cache redundancy, and unnecessary data evictions leading to increased miss ratios are the norm, and not the exception, especially in extremely large scale workloads. Hermes addresses these issues by maintaining global file heatmaps that represent how a file is accessed across processes or applications. It uses those heatmaps to express the placement of data in a hierarchical system.

The contributions of this work include:

- presenting the design and implementation of Hermes: a new, heterogeneous-aware, multi-tiered, dynamic, and distributed I/O buffering system (Section 3.1).
- introducing three novel data placement policies to efficiently utilize all layers of the new memory and storage hierarchy (Section 3.2.2).
- presenting the design and implementation of three novel techniques to perform *memory*, *metadata*, and *communication* management in hierarchical buffering systems (Section 3.4.2).
- showcasing that a data-centric prefetching approach solves several issues caused by a growing set of application-specific optimizations (Section 3.3).
- evaluating Hermes' design and technical innovations showing that our solution can grant better performance compared to the state-of-the-art buffering platforms (Section 4).

2 Background

2.1 Modern Application I/O Characteristics

Modern HPC applications are required to process large volume, velocity and variety of data, leading to an explosion of data requirements and complexity*. Many applications spend significant time of the overall execution in performing I/O making storage a vital component in performance [26]. Furthermore, scientific applications often demonstrate bursty I/O behavior [27, 28]. Typically, in HPC workloads, short, intensive, phases of I/O activities, such as checkpointing and restart, periodically occur between longer computation phases [29,

*http://www.hpcuserforum.com/presentations/tuscon2013/ IDCHPDABigDataHPC.pdf

30]. The intense and periodic nature of I/O operations stresses the underlying parallel file system and thus, stalls the application. To appreciate how important and challenging the I/O performance of a system is, one needs to deeply understand the I/O behavior of modern scientific applications. More and more scientific applications generate very large datasets, and the development of several disciplines greatly relies on the analysis of massive data. We highlight some scientific domains that are increasingly relying on High-Performance Data Analytics (HPDA), the new generation of data-intensive applications, which involve sufficient data volumes and algorithmic complexity to require HPC resources:

- *Computational Biology*: The National Center for Biotechnology Innovation maintains the GenBank database of nucleotide sequences, which doubles in size every 10 months. The database contains over 250 billion nucleotide bases from more than 150,000 distinct organisms.
- *Astronomy*: Square Kilometre Array project run by an international consortium operates the largest radio telescope in the world which produces staggering data as presented in the keynote speech during the 2017 SC conference. As highlighted, the incoming images are of 10 PBs and the produced 3D image is 1 PB each.
- *High-Energy Physics*: The Atlas experiment for the Large Hadron Collider at the Center for European Nuclear Research generates raw data at a rate of 2 PBs per second and stores approximately 100 PBs per year of processed data.

2.2 A New Memory and Storage Hierarchy

Accessing, storing, and processing data is of the utmost importance for the above applications which expect a certain set of features from the underlying storage systems: a) high I/O bandwidth, b) low latency, c) reliability, d) consistency, e) portability, and f) ease of use. New system designs that incorporate non-volatile buffers between the main memory and the disks are of particular relevance in mitigating the periodic burstiness of I/O. The new DMSH promises to offer a solution that can efficiently support scientific discovery in many ways: improved application reliability through faster checkpoint-restart, accelerated I/O performance for small transfers and analysis, fast temporary space for out-of-core computations and in-transit visualization and analysis. Building hierarchical storage systems is a cost-effective strategy to reduce the I/O latency of HPC applications. However, while DMSH systems offer higher I/O performance, data movement between the layers of the hierarchy is complex and significantly challenging to manage. Moreover, there is no software yet that addresses the challenges of DMSH.

Middleware layers, like MPI-IO and parallel HDF5, try to hide the complexity by performing coordinated I/O to shared files while encapsulating general purpose optimizations. However, the actual optimization strategy of these middleware layers is dependent on the underlying file system software and hardware implementation. More importantly, these middleware libraries are designed with memory-to-disk endpoints and are not ready to handle I/O access through a DMSH system, which is ultimately left to the user. Ideally, the presence of multiple layers of storage should be transparent to applications without having to sacrifice performance or increase programming difficulty. System software and a new middleware solution to manage these intermediate layers can help obtain superior I/O performance. Ultimately, the goal is to ensure that developers have a high-performance I/O solution that minimizes changes to their existing software stack, regardless of the under-

lying storage.

Deep memory and storage hierarchies require a scalable, reliable, and high-performance software to efficiently and transparently manage data movement. New data placement and flushing policies, memory and metadata management, and an efficient I/O communication fabric is required to address DMSH complexity and realize its potential. We believe that a radical departure from the existing software stack for the scientific communities is not realistic. Therefore, we propose to raise the level of abstraction by introducing a new middleware solution, Hermes, and make it easier for the user to perform I/O on top of a DMSH system. In fact, Hermes supports existing widely popular I/O libraries such as MPI-IO and HDF5 which makes our solution highly flexible and production-ready. We envision a buffering platform that can be application- and system-aware, and thus, hide lower level details allowing the user to focus on his/her algorithms. We strive for maximizing productivity, increasing resource utilization, abstracting data movement, maximizing performance, and supporting a wide range of scientific applications and domains.

2.3 Accelerating Read Access Time

Data prefetching is a well-understood data access optimization that has been explored in the literature throughout the years. Starting with hardware prefetchers [31 - 36], data moves through the main memory into the CPU caches to increase the hit ratio thereby increasing data locality. The granularity of a hardware prefetcher is a memory page, and the trigger is executed per-core. The hardware performs locality-aware prefetching (i.e., read-ahead approach) where once a memory page is accessed, the prefetcher brings the next page into the caches (temporal and spatial locality). The ability to detect strided patterns is also present in most modern CPU architectures*. However, if the application demonstrates irregular patterns, then the miss ratio is high, and applications experience performance degradation due to the contention in memory bus between the normal memory access and the prefetcher. Lastly, a memory page is a well defined prefetching unit while the same cannot be said for I/O where file operations will be variable-sized. Software-based solutions [20 -21, 37 - 41] leverage information collected from the application to perform data prefetching and can be broadly categorized into:

2.3.1 Offline Data Prefetchers

This category of prefetchers involve a pre-processing step where an analysis of the application determines the data access patterns and devise a prefetching plan. There are several different ways to perform this pre-execution analysis and several ways to devise a prefetching plan. In trace-driven [20] prefetching, the application runs once to collect execution and I/O traces. These are then analyzed to generate prefetching instructions. This method offers high accuracy in both the *when* and *what* to prefetch but requires significant user-involvement and poses large offline costs. More importantly, a trace-driven approach suffers from the fact that an application's I/O behavior is subject to change at runtime. For example, the applications may include third party libraries (which could result in a mismatch between application's I/O calls and what the servers experience) or when the application runs with different inputs than originally traced (which could result in a different access pattern). Similar to trace-driven approach, a history-based [38, 42] prefetcher stores the seen accesses in a previous run of an application into a database, and, thus, access patterns are

*https://tinyurl.com/lxxw7sn

known when the same application executes again in the future. While this method decreases the level of user involvement and the cost of trace analysis, it assumes that the application's behavior remains stable between executions (which is unrealistic). Another approach to identify application access patterns is compiler-based prefetching. In this method, the source code is analyzed and modified to add prefetching instructions either by I/O re-ordering [43] (where calls are moved earlier in the code) or by hint-generation [20] (where new code is injected) to provide the information to the prefetcher about when and what to prefetch. The code is then re-compiled and executed with the extra prefetching instructions. This approach avoids the increased offline costs, since it does not require any execution of the application, but only requires the modification of the source code (which raises security concerns). Moreover, it suffers from miscalculations as to how far up in the code it should move the I/O calls or inject the hints to perform the prefetching on-time. Lastly, data staging [44] is a form of prefetching by pre-loading the working set of an application (i.e., all data that will be read) in dedicated staging resources before the application even starts. This method leads to high hit ratios, but it assumes that the working set can fit in the staging resources capacity. In other words, it leads to sub-optimal resource utilization since data is kept in memory for the entirety of the application's runtime and may be subject to undesired evictions before the data is read.

2.3.2 Online Data Prefetchers

This category of prefetchers trade accuracy for a "learn as you go" model. The application's access patterns are learned as the execution proceeds, avoiding any pre-processing steps. The on-the-fly identification of data access patterns can be done using several models. *Firstly*, statistical methods such as hidden Markov models (HMM) [45, 46] and ARIMA models [47] require a large number of observations to accomplish model convergence. Once the model has converged, it can predict the next access and trigger prefetching. However, they often focus exclusively on either spatial or temporal I/O behaviors and need long execution time or several runs to achieve accurate predictions. *Secondly*, a grammar-based model [22, 48, 49] relies on the fact that I/O behaviors are relatively deterministic (inherent from the code structure) and predicts when and what future I/O operations will occur. However, this method demands repetitive workloads and does not work well for irregular access patterns since the grammar cannot be built out of randomness. *Lastly*, machine learning approaches [50, 51] have been recently proposed where a model learns the data access pattern and uses it to drive the prefetching plan. Rather than relying on a statistical distribution of accesses or a sequence of symbols, this method relies on a type of probabilistic language model called n-gram [52]. This model predicts the next item in a sequence which takes the form of a collection of subsequences of N consecutive tokens in a stream. All online approaches share the fact that they do not rely on a priori knowledge of the application's data access patterns or user inputs and hints. The problem is that they require a warm-up period at the beginning of the execution as they build their models, which can result in added overheads and low performance. Additionally, online prefetchers' performance is directly related to the predictive capabilities of the models used, causing accuracy and timeliness to be suboptimal when compared to the offline approaches.

The common theme for all existing approaches is that they implement a client-pull model. Prefetching is driven by the applications and their data access pat-

terns. As we move closer to exascale, supercomputers, while sharing prefetching resources, are expected to run multiple concurrent applications possibly connected in a workflow. This means that application-specific optimizations will not perform due to a lack of global coordination. Prefetching cache space will be limited and shared, leading to cache pollution, cache redundancy, and unwanted evictions. Application-bound prefetching resources will be competing with one another, leading to loss of performance due to interference. The complexity of modern workflows renders application-centric solutions unrealistic. We need to address the challenges of read-optimizations from a data-centric view. Systems must evolve and make smart decisions based on how data is accessed and what common patterns can be found across components of a workflow. A server-push model can obtain a global view and apply optimizations on the most valuable pieces of data. None of the existing data prefetching solutions fully utilize the hierarchical environment. The amount of RAM available to each core is shrinking, and the presence of additional layers of fast storage mediums seems like a natural solution to mitigate this issue. The hardware is there, but we need to design software to drive performance by masking access latency behind each tier of the DMSH.

3 Design and Implementation

3.1 Hermes Architecture

3.1.1 Design overview

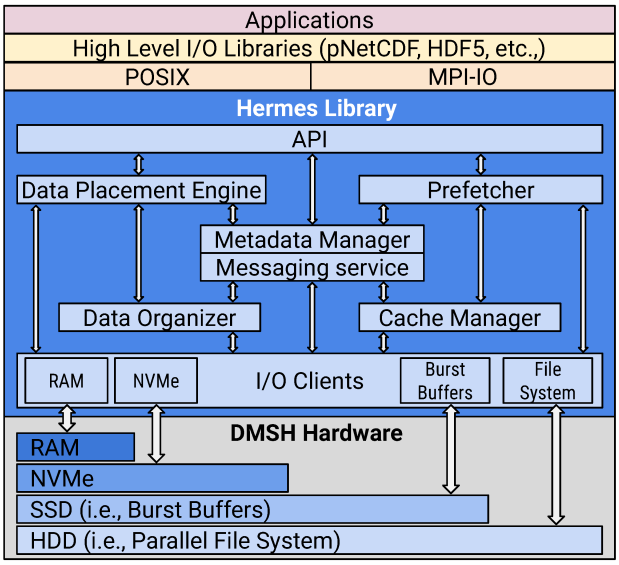

Fig.1. Software stack and Hermes internal design.

Hermes is designed as a middleware layer - sitting between applications and DMSH as shown in Figure 1. As a middleware library, Hermes captures I/O calls, both POSIX and HDF5 (i.e., fopen, fread, fwrite, and H5Fcreate, H5Dread etc.) and redirects them to different layers of DMSH. Legacy applications can easily connect to Hermes by simple linking (i.e., LD_PRELOAD) or recompiling the code with our library. There are no changes to user code and there is no need to upgrade to a different workflow. Similarly, there is no requirement to change anything on the underlying storage (i.e., Lustre, PVFS, etc.,) since Hermes services reside within the application. Hermes is a middleware software that does not require any modifications to existing runtime services. The lifetime of Hermes is tightly couple with that of the application that has linked to it. We design Hermes to easily work with existing software. Our goal is to maximize user productivity by making I/O buffering transparent.

As a separate usage mode, Hermes also provides a

new buffering API for users who want to explicitly take control of the data movement between layers of DMSH. This mode also allows Hermes to perform active buffering where data is shipped to the buffer nodes along with specific instructions or operations to be performed on them. For example, a user can pass a set of integers to Hermes instructing it to first store them to the buffer nodes, then sort them, compress the sorted list and lastly persist the final result to the remote PFS. This flow can be easily executed by a series of hinting mechanisms (i.e., flags) that Hermes provides to the user. Our hinting mechanism is a simple bit encryption which indicates predetermined operations like sorting, compression/decompression, deduplication and others. For user defined operations, Hermes provides a bootstrapping mechanism in which the user can submit his/her functions. The library will then compile and place the executables to a registry of operations to be handled by the buffering nodes. Reserved bits are used for user-defined operations.

The high-level architecture of Hermes can be seen in Figure 2. In DMSH systems, besides the main memory, every compute node might be equipped with an NVMe device or even an SSD. Additionally, shared buffering nodes, such as burst buffers, will most likely be present and positioned close to the compute nodes. Finally, a remote PFS supports all compute nodes with persistence and fault tolerance as important features. Hermes is a platform that aims to enable efficient access to the layers of DMSH and as such we distinguish two data paths: *a vertical and a horizontal hierarchy*. Vertical hierarchy refers to data movement within a compute node and all the way down to the burst buffers and PFS. Horizontal hierarchy refers to sending data to another compute node's RAM or NVMe device. The horizontal data movement is greatly optimized if there is an RDMA-capable network but Hermes can also support systems with no RDMA. Therefore, a DMSH system could consist of several layers, performance-wise, such as local RAM, remote RAM, local NVMe, remote NVMe, burst buffers, and PFS (numbered in fig. 2).

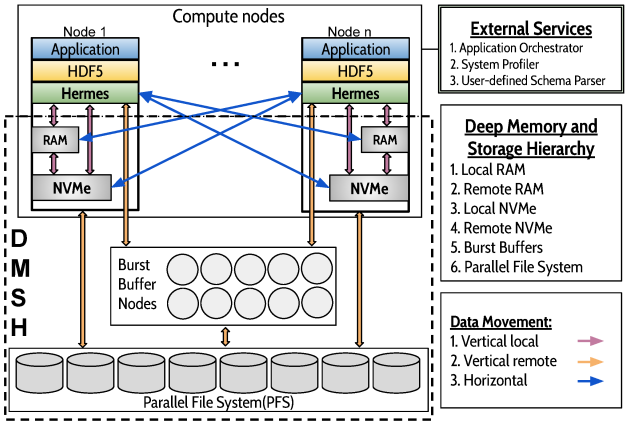

Fig.2. Hermes internal design.

3.1.2 *Internal components*

Figure 1 demonstrates the design of Hermes library and all the internal components that work together to achieve an efficient, transparent, and easy-to-use data access in all layers of a DMSH (i.e., both vertically and horizontally). The main Hermes library is complemented by a set of tools and services that help achieve broader goals such as multi-tenancy, adaptability, etc. Brief description of each component's responsibilities:

API: The API is responsible for intercepting all I/O calls from the applications. It also calculates the operations to be carried out by the buffering nodes in case of an active buffering scenario.

Data Placement Engine: This engine is responsible for mapping data onto DMSH. In other words, the data placement engine calculates the data destination, where in the hierarchy should the data be redirected. It maps data according to various data placement policies.

Data Organizer: The main responsibility of this component is to move data between the layers of DMSH. It is triggered by other components according to certain criteria which makes it an event-based component.

For instance, if there is no space left in NVMe, data organizer is triggered to move data down to the burst buffers and thus freeing space in NVMe. This component is responsible for carrying out all data movement either for prefetching reasons, evictions, lack of space, or hotness of data etc.

Metadata Manager: The MDM maintains two types of metadata information: user's and Hermes library's internal metadata. Since Hermes can transparently buffer data by intercepting I/O calls, MDM keeps track of user's metadata operations (i.e., files, directories, permissions etc.) while consulting the underlying PFS. Additionally, since data can be buffered anywhere in the hierarchy, MDM tracks the locations of all buffered data and internal temporary files that contain user files.

Cache manager: This component is responsible for handling all buffers inside Hermes. It is equipped with several cache replacement policies such as least recently used (LRU) and least frequently used (LFU). It works in conjunction with the prefetcher. It can be configured to hold "hot" data for better I/O latency. It is also responsible for implementing application-aware caching schemas.

Prefetcher: This component is performance-driven. It implements a server-push model that can achieve better data-prefetching performance by leveraging a global view of how data are accessed across multiple applications. The details of this component are presented in details in Subsection 3.3.

Messaging Service: This component is used to pass small messages across the cluster of compute nodes. This component does not involve any data movement which is actually done by either the application cores or other Hermes components such as the data organizer and prefetcher. Instead, this component provides an infrastructure to pass instructions to other nodes to perform operations on data or facilitate its movement. For example, a typical type of message in Hermes is to flush buffered data of a certain file to the next layer or to PFS.

I/O Clients: These clients refer to simple calls using the appropriate API based on the layer of the hierarchy. For instance, if Hermes data placement engine maps some data to the burst buffers, then the respective I/O client will be called and perform the fwrite() call. Internally, Hermes can use POSIX, MPI-IO, or HDF5 to perform the I/O. An important feature of Hermes is that user's data structures are mapped to Hermes' internal structures at each layer of DMSH. For example, an original dataset of an HDF5 file could be mapped into a temporary POSIX file in NVMe. The I/O clients give Hermes the flexibility to "talk" to several data destinations and manage the independent systems (e.g., memcpy for RAM, fwrite() for NVMe, MPI_File_write() for burst buffers).

System Profiler: This component is a service outside the main library. It is designed to run once during the initialization. It performs a profiling of the underlying system in terms of hardware resources. It tries to detect the availability of DMSH and measure each layer's respective performance. It is crucial to identify the parameters that Hermes needs to be configured with. Using this information, the data placement engine can do a better job when mapping data to different layers. Each system will have different hierarchy. Additionally, each hierarchy will demonstrate different performance characteristics. In our prototype implementation this component is external and results are manually injected to the configuration of the library. We plan to automate this process.

Schema Parser: This component accepts a user-defined buffering schema and embeds it into the library. This schema is passed in a XML format and Hermes is configured accordingly. For instance, if user chooses to

aggressively buffer a certain dataset or file, then Hermes will prioritize this data higher up in the hierarchy and also the cache manager will get informed not to evict this specific buffered dataset. All this is possible because Hermes will use the user's instructions to offer the best buffering performance. In our prototype implementation schema parser is external and is planned to be automated in future versions of Hermes.

Applications Coordinator: This component is designed to offer support in a multiple-application environment. It manages the access to the shared layers of the hierarchy such as the burst buffers. Its goal is to minimize interference between different applications sharing this layer. Additionally, it coordinates the flushing of the buffers to achieve maximum I/O performance. More information on this component can be found in [25].

All the above components allow Hermes to offer a high performance I/O buffering platform which is highly configurable, easily pluggable to several applications, adaptable to certain system architectures, and feature-rich yet lightweight.

3.2 Hermes Buffering Modes and Policies

3.2.1 Buffering modes

Similar to other buffering systems, Hermes offers several buffering modes (i.e., configurable by the user) to cover a wide range of different application needs such as I/O latency, fault tolerance, and data sharing:

A. Persistent: in this mode, data buffered in Hermes is also written to the PFS for permanent storage. We have designed two configurations for this mode. 1) Synchronous: directs write I/O onto DMSH and also to the underlying permanent storage before confirming I/O completion to the client. This configuration is designed for uses cases such as *write-though* cache or *stage-in* for read operations. Since all data also exist in the PFS, synchronous-persistent mode is highly fault-tolerant, offers strong data consistency, is ideal for data sharing between processes, and supports read-after-write workloads. However, it demonstrates the highest latency and lowest bandwidth for write operations since data directed to the buffers also need to be written in the PFS. 2) Asynchronous: directs write I/O onto DMSH and completion is immediately confirmed to the client. The contents of buffers are eventually written down to the permanent storage system. The trigger to flush buffered data is configurable and can be: i) per-operation, flushing is triggered at the end of current fwrite(), it also flushes all outstanding previous operations, ii) per-file, flushing is triggered upon calling fclose() of a given file (this is similar to Data Elevator approach), iii) on-exit, flushing is triggered upon application exit (this is similar to Datawarp approach), and iv) periodic, flushing is periodically triggered in the background (this is the default Hermes setting). This configuration is designed for use cases such as *write-back* cache and *stage-out* for read operations. It provides low-latency and high bandwidth to the application since processes return immediately after writing to the buffers. It also offers eventual consistency since data are flushed down eventually. It is ideal for write-heavy workloads and out-of-core computations.

B. Non-persistent: in this mode, I/O is directed to DMSH and is never written down to the permanent storage. It is designed to offer a scratch space for fast temporary I/O. Upon application exit, Hermes deletes all buffered data. This mode can be used for scenarios such as quickly storing intermediate results, communication between processes, in-situ analysis and visualization. In case of buffering node failures, application must restart. This mode offers high bandwidth and low latency. Lastly, applications can reserve a specific allocation (i.e., capacity on buffers) for which data preser-

vation is guaranteed by Hermes (similar to Datawarp reservations). These allocations expire with the application lifetime. In case of buffer overflow, Hermes will transparently swap buffer contents to the PFS much like memory pages are swapped to the disk by the OS. The mechanism was designed to offer some extra degree of flexibility to Hermes. For example, let us assume that an application writes simulation results every 5 minutes. These results are directly read from the buffers by an analysis kernel which writes the final result to the PFS for permanent storage. Simulation data can be deleted or overwritten after the analysis is done. Hermes can utilize this periodic and bursty I/O behavior and write the next iteration on top of the previous one instead of wasting extra buffer space. To achieve this conditional overwriting of data, Hermes utilizes a flagging system to define the lifetime of buffered data.

C. Bypass: in this mode, as the name suggests, I/O is performed directly against the PFS effectively bypassing Hermes. This mode resembles *write-around* cache designs.

3.2.2 Data placement policies

In DMSH systems, I/O can be buffered to one or more layers of the hierarchy. There are two main challenges: i) how and where in the hierarchy data are placed, ii) how and when do buffers get flushed either in the next layer or all the way down to PFS. In Hermes, the first challenge is addressed by the data placement engine (DPE) component and the second by the data organizer. We designed four different data placement policies to cover a wide variety of applications' I/O access patterns. Each policy is described by a dynamic programming optimization* and follows the flow of Algorithm 1.

Algorithm 1: Hermes DPE algorithm to calculate data placement in DMSH (pseudo code)

```
Procedure DPE(data, tier)
    if data can fit in tier then
        /* buffer data in this layer    */
        PlaceData(data, tier)
    else
        /* buffer in next tier          */
        p1 ← DPE(data, tier.next)
        /* split data based on the
           remaining capacity of the
           current tier                 */
        data[] = Split(data,tier)
        p2 ← DPE(data[0], tier) +
          DPE(data[1], tier.next)
        /* flush current tier to create
           space and then place data    */
        p3 ←
          Flush(data, tier) + DPE(data, tier)
        max(p1, p2, p3)
    end
```

The general idea of the algorithm is as follows. First, if the incoming data can fit in the current layer's remaining capacity, it places the data there (i.e., *PlaceData(data, tier)*). In case it does not fit, based on the constraint of each policy, it tries one of the following: a) solve again for next layer (i.e., *DPE(data, tier.next)*), b) place as much data as possible in the current layer and the rest in next (i.e., *DPE(data[0], tier) + DPE(data[1], tier.next)*), and c) flush current layer and then place new incoming I/O (i.e., *Flush(data, tier) + DPE(data, tier)*). We implemented the DP algorithm using memoization techniques to minimize the overhead of the solution. We further provide a configuration knob to tune the granularity of triggering the optimization code for data placement.

A. Maximum Application Bandwidth (MaxBW): this policy aims to maximize the bandwidth applications experience when accessing Hermes. The DPE places data in the highest possible layer of DMSH in a top-down approach, starting from RAM,

*Full mathematical formulation of each policy can be found in the Appendix.

while balancing bandwidth, latency, and the capacity of each layer. For instance, this policy will place incoming I/O into RAM, if the data size fits in RAM's remaining capacity. Otherwise, the policy will try to minimize the I/O time between the following actions: skip RAM and directly place data in NVMe, place as much data as RAM can hold and the rest in NVMe, or first create space in RAM by flushing data in LRU fashion and then place all new data in RAM. The approach applies to all layers making the solution recursively optimal in nature. The above data placement policy is expressed as an optimization problem where DPE minimizes the time taken to write the I/O in the current layer and the access latency to serve the request, effectively maximizing the bandwidth. The data organizer moves data down periodically (or when triggered) to increase the available space in upper layers for future incoming I/O. Data movement between layers is performed asynchronously. This policy is the default Hermes configuration.

B. Maximum Data Locality: this policy aims to maximize buffer utilization by simultaneously directing I/O to the entire DMSH. The DPE divides and places data to all layers of the hierarchy based on a data dispersion unit (e.g., chunks in HDF5, files in POSIX and independent MPI-IO, and portions of a file in collective MPI-IO). Furthermore, Hermes maintains a threshold based on the capacity ratio between the layers of the hierarchy. This ratio reflects on the relationship between each layer (e.g., system equipped with 32GB RAM, 512GB NVMe, and 2TB burst buffers creates a capacity ratio of 1-16-64). The data placement in this policy accounts for both layer's capacity and data's spatial locality. For instance, this policy will place incoming I/O in RAM if its data size fits within the capacity threshold while respecting the locality of the file. If it does not fit in RAM's remaining capacity, the DPE will try to maximize the buffer utilization between the following actions: skip RAM and place data in NVMe, place as much data as possible in RAM and the rest in NVMe, or perform a re-organization of the files and thus, creating space for the new data in RAM. The above process is recursive and can be expressed as an optimization problem. DPE minimizes the time taken to write the I/O in the current layer and the degree of data dispersion (i.e., how many layers data are placed to) effectively maximizing the buffer utilization. Data movement between layers is performed asynchronously. This policy is ideal for workflows that encapsulate partitioned I/O. For instance, one could prioritize a certain group of MPI ranks over another (e.g., aggregator ranks) or one type of file over another (e.g., metadata files over data files).

C. Hot-data: this policy aims to offer applications a fast cache for frequently accessed data (i.e., hot-data). The DPE places data in the hierarchy based on a hotness score that Hermes maintains for each file. This score encapsulates the access frequency of a file. Highest scored files will be placed higher up in DMSH since they are expected to be accessed more often. This ensures that layers with lower latency and higher bandwidth will serve critical data such as metadata, index files, etc. The DPE also considers the overall file size to efficiently map data to each layer (i.e., smaller files buffered in RAM whereas larger files in burst buffers). The data placement policy can be expressed as an optimization problem where DPE minimizes the time taken to write the I/O in the current layer considering both hotness and capacity of layers. The data organizer demotes or promotes data based on the hotness score and the data movement is performed asynchronously. This policy is ideal for workflows that demonstrate a spectrum of hot-cold data.

D. User-defined: this policy aims to support user-

defined buffering schemas. Users are expected to submit an XML file with their preferred buffering requirements. This file is parsed during initialization by the schema parser component and used by the DPE to make data placement decisions. For instance, user can define certain files to always be in RAM (i.e., never get evicted), or which HDF5 chunks to get buffered in NVMe etc.

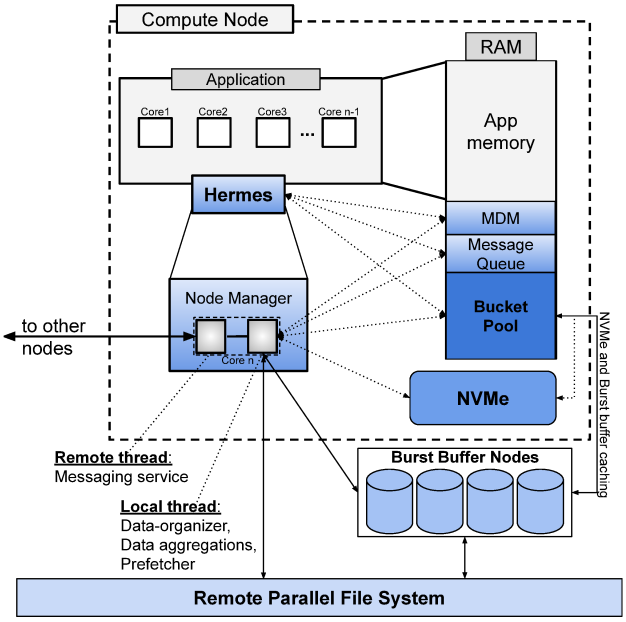

Fig.3. Hermes Prefetching Design.

3.3 Data-centric Multi-tiered Prefethings

Hermes implements a system-wide, data-centric, server-push prefetching solution that aims to identify how files are accessed, regardless of which process or application does the data access, and utilize this information to pre-load the data needed into the deep memory and storage hierarchy. Hermes' Prefetcher optimizes read operations by leveraging two main observations: a) the presence of multi-tiered storage suggests a feasible solution to the shrinking DRAM size per-core; a prefetching solution that utilizes the storage hierarchy to fetch data in a pipeline fashion is needed, and, b) identifying an application's data access pattern will not suffice in the age of data-intensive computing; a global view of how files are accessed across a workflow is needed to place prefetched data at the right tier of the hierarchy (i.e., hotter data are fetched to the higher, more capable, levels of the hierarchy such as memory). Figure 3 shows the design of Hermes Prefetcher.

The main idea is to fetch portions of a file to a tier of the hierarchy based on access frequency, recency, and relationship between segments (i.e., file segment sequencing). In other words, instead of guessing what an application will access next, Hermes collects access statistics of file regions (which we call file segments) from the file systems themselves and pro-actively loads them in the hierarchy, based on a segment score, that reflects the urgency to access the chosen segment. This score basically incorporates the frequency with which the segment is accessed across processes or applications thereby creating a file access heatmap. The file heatmap is then used to naturally match it to a hierarchical environment. Segment movement between tiers of the hierarchy is also based on how recently a segment was accessed. In effect, Hermes' prefetcher answers the three prefetching questions (what to prefetch, when to prefetch, and where to place prefetched data) indirectly by naturally mapping the spectrum of segment frequency and recency to the appropriate tier leveraging the hardware capabilities of each tier. Hermes' prefetcher aims to optimize complex scientific workflows where a collection

of data producers (i.e., simulations, static data sources, etc.) send data down a pipeline and a collection of consumers (i.e., analytics, visualization) process the data multiple times. Our design fits naturally in such environment with hierarchical data prefetching boosting read operations across all data consumers.

Hermes' Prefetcher follows an event-based paradigm. Each compute node is equipped with an Hermes node manager running on one of the cores. Each application dynamically links to the Hermes library and a background prefetching thread, we call *Agent*, is spawned alongside each application process. Upon application initialization (e.g., $MPI\_Init()$), a small fraction of the main memory is allocated for prefetcher internal structures. Hermes' Prefetcher is a multi-threaded program consisting of the following components:

- Hardware Monitor: Its main role is to monitor all available hardware tiers. The events are generated by the system and are pushed to an in-memory event queue which is served by a pool of daemon threads. In this context, events are either file accesses or tier remaining capacity. All collected events are then passed on to the file segment auditor. In the face of updates events, the prefetcher invalidates the previously prefetched data enforcing data consistency.

- File Segment Auditor: Its main role is to calculate file segment statistics. Specifically, for each file segment, which is practically a region of a file, the auditor calculates its access frequency, when was it last accessed, and which segment access preceded it. Using this information, the auditor can construct a score for each file segment that reflects how hot the segment is in the prefetching context. A hot segment is one that is accessed many times in a recent time window. The sequencing of segments also provides a logical map of which segments are connected to one another. Lastly, segment statistics and mappings are both maintained in the metadata manager.

- Agent Manager: Its main role is to collect the beginning and the end of a prefetching epoch enclosed between a file open and file close calls and pass it to the auditor who marks the appropriate file segments that are targeted for prefetching. The Agent is able to intercept POSIX, MPI-IO, and HDF5 open-close calls.

3.3.1 File Segment Scoring

A *file segment* is defined as a file region enclosed by start and end offsets. The segment size can be statically defined (e.g., every 1MB) or it can be dynamic based on how the file is being read. A file segment is the prefetching unit within the prefetcher, which means all prefetching operations are expressed by loading one or more segments. Its dynamic nature provides Hermes a better opportunity to decompose read accesses in finer granularity and better utilize the available prefetching cache, especially in a hierarchical environment where the prefetching cache can span multiple tiers. Each incoming read request may correspond to one or more segments. For example, assume the segment size is 1MB and there is an fread() operation starting at offset 0 with 3MB size, then Hermes will prefetch segments 1, 2, and 3 to optimize this data access. For every segment, Hermes' Prefetcher maintains its access frequency within a prefetching epoch, when it was last accessed, as well as which segment preceded it (i.e., segment sequencing). It scores each file segment based on these collected access statistics by the following formula:

$$Score_s = \sum_{i=1}^{k} (\frac{1}{p})^{\frac{1}{n}*(t-t_i)} \quad (1)$$

where s is the segment being scored, k is the number of accesses, t is the current time, t_i is the time of the i^{th} access, and $n \geq 1$ is the count of references to segment s. An intuitive meaning of $\frac{1}{n}$ is that a segment's score is reduced to $\frac{1}{p}$ of the original value after every time step. Finally $p \geq 2$ is a monotonically non-increasing class of functions. Consistently with the principle of temporal locality, $t - t_i$ gives more weight to more recent references with smaller backward distances. This score aims to encapsulate three simple observations about the probability of a segment being accessed in the future. A segment is likely to be accessed in the future again if: a) it is accessed frequently, b) it has been accessed recently, and c) it has multiple references to it. All calculated segment scores are also kept as an index in an ordered map to avoid excessive sorting of score values.

The file heatmaps are generated by the score of each segment. To minimize overheads, the auditor maintains segment statistics and file heatmaps in the MDM for the duration of an epoch (i.e., while the file remains opened for read). Upon closing the file Hermes has the ability to store the file heatmaps on disk resembling a file access history. When a file gets re-opened, if there is a stored heatmap, Hermes will load it in memory and compare observed accesses with the pre-existing heatmap. New accesses will evolve the heatmap further. Heatmaps get deleted once the workflow ends (i.e., a collection of simulation - analysis programs executed in a pipeline). .

3.3.2 Prefetched Data Placement

Hermes' Prefetcher is a truly hierarchical data prefetcher, and thus, the prefetching cache spans across multiple tiers of the deep memory and storage hierarchy. In contrast to existing prefetching solution where prefetched data have a single destination, the main memory, Hermes fetches data into multiple tiers using its data placement engine and the collected segment statistics. This approach can lead to better resource utilization, masking access latency behind each tier (i.e., while prefetching segments in RAM, other segments are also prefetched to higher tiers), and can offer concurrent access with less interference (i.e., while one application accesses segments from RAM another one can access from NVMe).

Algorithm 2: Hermes Prefetching algorithm data placement in DMSH (pseudo code)

```
Procedure Place(segment, tier)
    if segment.score > tier.min_score then
        if segment cannot fit in this tier then
            tier.min_score ← segment.score
            DemoteSegments(segment.score,tier)
        end
        if segment.score > tier.max_score
         then
            tier.max_score ← segment.score
        end
        place segment in this tier
    else
        Place(segment, tier.next)
    end
Procedure DemoteSegments(score, tier)
    segments ←
     GetSegmentsLowerThan(score, tier)
     foreach s ∈ segments do
        Place(s, tier.next)
    end
```

DPE periodically monitors the segment score changes from the auditor to decide if and what segments should be moved up or down the tiers. All updated scores are pushed by the auditor into a vector which the engine processes. To avoid excessive data movements among the tiers, Hermes uses two user-configurable conditions to trigger the prefetching triggers: a) a time interval (e.g., every 1 sec), and, b) a number of score changes (e.g., every 100 updated scores). The engine maintains a min and max segment score for each available tier. If an updated segment score violates its current tier placement, then it gets promoted or demoted accordingly. This approach also handles automatic evictions since each segment has its natural po-

sition in the hierarchy based on its score. Note, if segments have exactly the same score, the default policy in Hermes is to randomly place them in the tiers. Algorithm 2 has a time complexity of $O(m * n)$ where m is the number of segments updated on that node and n is the number of layers. Note, $n << m$ and m is the number of segments updated on a node between an interval t. This t should be ideally configured close to the average computation time of all the application in the workflow, to avoid excessive computations.

3.4 Implementation Details

3.4.1 Node design

The new DMSH system architecture suggests that compute nodes may be equipped with one or more non-volatile storage device and share access to a burst buffer deployment. Hermes is designed to support all the new trends in system design. Each application core uses an I/O API (i.e., POSIX, MPI-IO, HDF5 etc.) which in turn is captured by Hermes. A dedicated core per node, called *Node Manager*, is exclusively used by Hermes services. Specifically, this multi-threaded core is responsible for metadata management, data organization and movement between layers, messaging services between compute nodes (horizontal hierarchy), local memory management such as placement of data in buckets, eviction policies, and finally prefetching. The ratio between application cores and the Hermes node manager is configurable and is suggested to be around 64-to-1 (i.e., similar to I/O forwarding layer present in several supercomputing sites). If an I/O forwarding layer exists, Hermes can utilize the I/O cores there. However, our design is not limited only to such systems and can be widely deployed. Figure 4 demonstrates Hermes node design. This design allows Hermes to maintain minimal overheads resulting to negligible impact to the application CPU and memory resources. In fact, Hermes' usage pattern on each node is not different from any other distributed storage service currently in production. For example, existing buffering solutions, such as Datawarp and DataElevator, run daemons on every node that are responsible to buffer incoming requests. Similarly, Hermes node manager is spawned on each node without affecting the application resources.

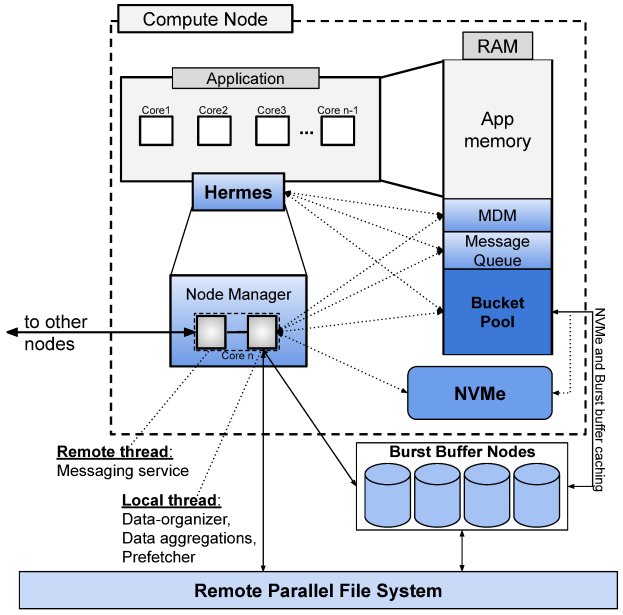

Fig.4. Compute node design in Hermes.

3.4.2 Critical components

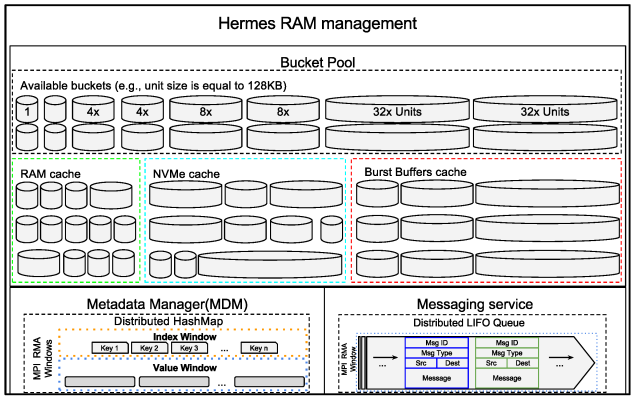

Fig.5. Hermes memory management.

During I/O buffering into DMSH, there are three critical operations: memory, metadata, and communication management. To achieve high-performance in each of

these critical operations, Hermes incorporates several novel technical innovations. As it can be seen in Figure 4, RAM is split into application memory and Hermes memory, which is further divided in bucket pool, MDM, and message queue. Each of these memory sections are further depicted in Figure 5.

A. RAM management. We have designed a new memory management system to offer fast and efficient use of main memory, a very crucial resource in any buffering platform. Hermes stores data in buckets, an abstract notion of a data holder. Buckets have a configurable fixed size and consist of a collection of memory pages. All buckets are allocated during the bootstrapping of the system, creating a bucket pool. This allows Hermes to avoid the cost of per-request memory allocation (i.e., only pay the cost in the beginning before application starts), to better control memory usage by avoiding expensive garbage collection, and to define the lifetime of memory allocations per application (i.e., re-use the same buckets after data have been flushed down). Bucket pools are organized in four regions: available buckets, RAM cache, NVMe cache, burst buffers cache. The bucket pool is managed by the bucket manager who is responsible to keep track of the status of each bucket (e.g., full - available). In the beginning, all buckets are available. When a process wants to buffer data, it asks the bucket manager for one or more buckets. The bucket, as a unit of buffering, is extremely critical to achieve high performance, low latency, and increases design flexibility (e.g., better eviction policies, hot data cache etc.).

We implemented Hermes' memory management using MPI one-sided operations. Specifically, buckets are placed in a shared dynamic Remote Memory Access (RMA) window. This allows easier access to the buckets from any compute node and a better global memory management. MPI-RMA implementations support RDMA-capable networks which further diminishes the CPU overhead. All bucket operations are performed by the manager who maintains an index of the entire RMA window and is responsible to assign buckets by returning a pointer to the application (i.e., to buffer data) or the data organizer (i.e., to flush data). Access to buckets occurs using *MPI_Put()* and *MPI_Get()*. Update operations are atomic with exclusive locking only on the bucket being updated. To support fast querying (e.g., location of a bucket, list of available buckets, etc.) the bucket manager indexes the RMA window and bucket relationships much like how *inode* tables work. The structure of a bucket includes an identifier (uint32), a data pointer (void*), and a pointer (uint32) to the next bucket. Hermes' buckets are perfectly aligned with RAM's memory pages which optimizes performance especially for applications with unaligned accesses. Finally, to ensure data consistency and fault tolerance, Hermes maps (via *mmap()*) the entire MPI-RMA window and the index structure to a file stored in a non-volatile layer of the hierarchy (configured by user). We suggest for a good balance of performance and safety against failures to place this special file to the burst buffers since if a compute node fails, the local NVMe device will become unavailable till the node is fixed.

Figure 6 motivates our design for Hermes' memory management. In this test, we issued a million fwrites of various sizes (from 64KB to 2MB) and measured the achieved memory operations per second. The test was conducted on our development machine that runs CentOS 7.1. In the test's baseline, we intercept each fwrite(), allocate a memory buffer (i.e., *malloc()*), copy data from user's buffer to the newly allocated space (i.e., *memcpy()*), and finally flush the buffer (i.e., *free()*) once the data are written to the disk. As a slightly optimized baseline case we used Google's TC Malloc. In contrast, Hermes intercepts each fwrite(), calculates

how many buckets are required to store the data and asks the bucket manager for them, and copies data from user's buffer to the acquired buckets. Once data are written to the disk, buckets are marked by the data organizer as available and no freeing is performed. As it can be seen in Figure 6, Hermes outperforms Linux's *Malloc* by 3x and *TCMalloc* by 2x. Hermes managed to sustain more than 3 million memory ops/sec, whereas the baselines, 1 and 2 million ops/sec respectively. Interestingly, as the allocation size grows, Linux's *Malloc* struggles in performance compared to *TCMalloc*. The pre-allocation and efficient management of the buckets and the lack of freeing of buffers helped Hermes to maintain stable high performance.

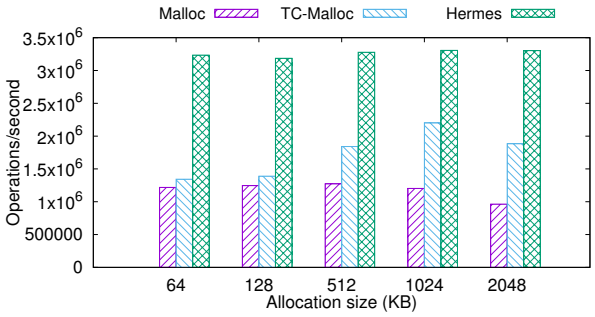

Fig.6. RAM operations throughput.

B. Metadata management. Any metadata service in distributed systems is subject to scalability and performance issues. Metadata in a buffering platform like Hermes consist of data distribution information (e.g., which node, which layer in DMSH, which bucket, etc.) and maintenance of both user's and internal file namespaces. Hermes' metadata manager is distributed and aims to offer highly concurrent and asynchronous operations. To achieve this, Hermes employs a novel distributed hashmap design, implemented using RMA windows and MPI one-sided operations. A hashmap consists of keys that correspond to specific values. Our design uses two RMA windows: i) *key window*, which is indexed to support efficient querying and ii) *value window*, for data values. This practically allows any process to simply *MPI_Get()* a specific key and then fetch its respective value. We use a 2-way hashing: first, the key is hashed to a specific node and then into a value that resides on that node. The MPI one-sided operations allow Hermes to perform metadata operations without interrupting the destination node. RDMA-capable machines will be able to perform even faster by using the RDMA controller for any data movement. Additionally, the RMA windows are dynamic effectively allowing the metadata to grow in size as required, similarly with rehashing in traditional hashmap containers. Lastly, our hashmap design liberates us to use complex structures, such as objects and nested custom datatypes, to describe a certain file and its metadata information. In contrast, popular in-memory key-value such as Redis or MemCached use simple datatypes for keys and values (e.g., strings or integers) which can be a limiting factor to metadata services. Additionally, these key-value stores offer features that are not useful in our use case such as replication, timestamps, and other features that only add overhead if one does not need or intend to use them.

Hermes' MDM uses several maps: i) file handler to file: maintains file handlers of opened files, {*fh,filename*}, ii) file to metadata properties: maintains all typical file properties (e.g., permissions, ownership, timestamps etc.,), {*filename,*{*filestat*}}, iii) files to location in DMSH: maintains data distribution information, {*filename,*{*(offset, size),(node,layer,type,identifier,freq)*}}, and iv) node to current status: maintains information for each node's current status such as remaining capacity, hot data access frequencies, etc., {*node,(layer,size,...)*}. These maps allow fast queries and O(1) read/write MDM operations without the need to execute separate services (e.g., a memcached server). Creation and up-

date of metadata information is performed by using MPI_EXCLUSIVE locks which ensures FIFO consistency. Read operations use a shared lock which offers higher performance and concurrency. Finally, Hermes' MDM exposes a simple and clean API to access its structures (e.g., `mdm_update_on_open()`, `mdm_get_file_stat()`, `mdm_sync_meta()`, etc.,).

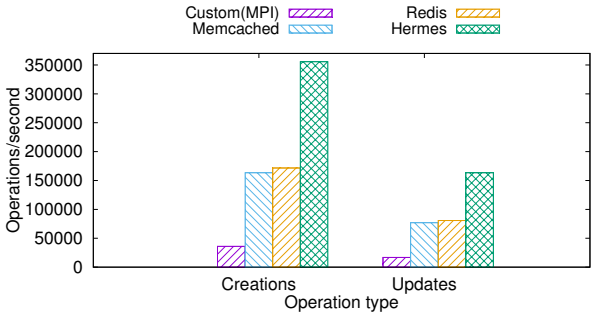

Fig.7. Metadata Manager throughput.

In Figure 7 we compare Hermes' MDM performance with a custom MPI-based solution, Memcached, and Redis. In this test, we issue a million metadata operations and we measure the MDM throughput in operations per second. First, we implemented a custom MPI-based solution where one process per node is the MDM and answers queries from other processes. Upon receiving one, it queues the operation, it spawns a thread to serve the operation, and it goes back to listening. The spawned thread removes the operation from the queue and performs the operation. While this approach is feasible, it uses a dedicated core per node. Another approach is to use an in-memory key-value store. We implemented the MDM using Memcached and Redis, two of the most popular solutions. In this approach, one memcached or Redis server per node is always running and awaits for any metadata operations. There is no explicit queuing but its implementation uses multi-threaded servers with locks and internal queues to support concurrent operations. Again, a dedicated core is required to run the server. Lastly, Hermes is using our own hashmap to perform metadata operations. Each processes accesses the shared RMA window to get or put metadata. There is no dedicated core used. As it can be seen in Figure 7, our solution outperforms by more than 7x the MPI-based custom solution and by more than 2x the Memcached and Redis versions. Update operations are more expensive since clients first need to retrieve the metadata, update them, and then push them back to the MDM.

C. Messaging service. Many operations in Hermes involve communication between different compute nodes, buffering nodes, and several other components. The messaging service does not involve in data movement but instead provides the infrastructure to pass instructions between nodes. For instance, horizontal access to the deep memory hierarchy involves sending data across the network to a remote RAM or NVMe. Another example is when the prefetcher gets triggered by one process it will fetch data to a layer of the hierarchy for subsequent read operations. Finally, when the buffers are flushed to the remote parallel file system for persistence, a system-wide coordination is required. All the above cases, require a high-performance and low latency messaging service to be in place. Hermes implements such messaging service by utilizing our own distributed queue via MPI one-sided operations. We designed a scalable messaging service by leveraging the asynchronicity of MPI RMA operations. When a process needs to communicate with another process across the compute nodes, it simply puts a message into the distributed queue that is hosted by all compute nodes. An shared dynamic RMA window is used to hold the queue messages. Each message has a type (i.e., an instruction to be carried out), its associated attributes, and a priority. As with the distributed hashmap above, if there is an RDMA controller it will be used to avoid interrupting the destination core. There is no need

to employ listeners or other always-on services such as Apache ActiveMQ [53] or Kafka [54] leading to better resource utilization. Additionally, we define our own bit encoding to keep the messages small and avoid costly serializations/transformations and therefore lead to lower latencies and higher throughput. Hermes messaging service aims to offer higher overall performance avoiding network bottlenecks and communication storms.

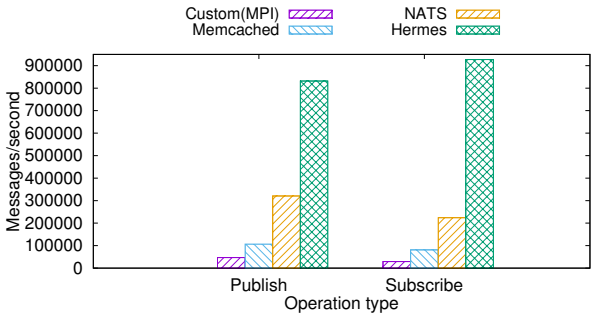

Fig.8. Messaging Service throughput.

In Figure 8 we compare Hermes' performance with a custom MPI-based solution, Memcached, and NATS. In this test, we issue a million queue operations (e.g., publish - subscribe) and we measure the messaging rate in messages per second. As described above, we implemented a custom MPI-based solution where one process per node accepts messages from other processes. We also implemented a distributed queue using Memcached where each message becomes a key-value pair (i.e., ID-message). Furthermore, we explored NATS, a popular, in-memory, high-performance, and open source messaging system. In both latter options, a dedicated core needs to run server code. Lastly, Hermes is using our own distributed priority queue to execute the messaging service. Each processes puts or gets messages from the shared RMA window while no dedicated core is used. As it can be seen in Figure 8, Hermes outperforms the custom MPI-based messaging implementation by more than 12x. This is expected since the server process gets saturated from the overwhelming rate of incoming messages. As a result, client processes needs to wait blocked for the server to accept their message. The handler thread cannot match the rate of new messages. A similar picture is evident in the memcached solution where Hermes performs more than 8x faster. However, in memcached, up to 4 handler threads are spawned which possibly leads to better performance compared to the custom MPI-based one. Finally, NATS performance is really good with more than 300000 published messages per second. However, Hermes outperforms NATS by more than 2x for publishing and more than 3x for subscribe operations.

3.5 Design Considerations

In this subsection, we briefly discuss concerns regarding the design and features of any buffering platform, especially one that supports a DMSH system such as Hermes. The goal is to present some of our ideas and to generate discussion for future directions.

A. High-performance:

Concern 1: How to support and manage heterogeneous hardware?

Hermes is aware of the heterogeneity of the underlying resources via the system profiler component which identifies and benchmarks all layers present in the system. Hermes aims to utilize each hardware resource to its best of its capabilities by avoiding hurtful workloads. Instead, Hermes' I/O clients generate access patterns favorable to the each medium.

Concern 2: How to avoid excessive network traffic?

Hermes' messaging service is carefully designed to operate with small-sized messages with bit encoding. Furthermore, by using asynchronicity and RDMA capable hardware our solution ensures the low network overhead.

Concern 3: How to support low-latency applications?

The several data placement policies of Hermes' DPE provide tunable performance guarantees for a variety

of workloads. For low latency applications, Hermes can leverage the performance characteristics of each layer by placing data to the fastest possible layer. Additionally, our novel memory management ensures that data can be efficiently cached in RAM before ending up to their buffer.

Concern 4: How to avoid possible buffer overflow?

Hermes' Data Organizer component manages the capacities of the layers and moves data up and down the hierarchy (i.e., between the layers). In corner cases of overflow, Hermes provides explicit triggers to the data organizer to re-balance the layers and move data based on the buffer capacity on each layer.

Concern 5: How to scale the buffer capacity?

Hermes' DPE can place data in remote RAM and NVMe devices, and thus, scaling is horizontal by adding more compute nodes. Additionally, Hermes can support RAM Area Network (RAN) deployments [55] to further extend the buffer capacity.

B. Fault tolerance:

Fault tolerance guarantees are based on the buffering mode selected (i.e., sync, async). In case of asynchronous buffering mode, buffered data are written to a fault tolerant layer such as a PFS eventually which means for a small window of time buffer contents are susceptible to failures. In our prototype implementation, buffers are flushed based on an event-driven architecture and also periodically to decrease the possibilities of losing critical data. As a future step, we want to investigate the following options: i) Checkpointing with configurable frequency. ii) Random replication per write operation. iii) DPE skips the failing component for incoming I/O.

C. Data consistency:

Concern 1: Data consistency model?

Hermes supports strong consistency for the application since our design avoids having the same buffered data in multiple locations and copies. Once a write is complete, any other process can read the data via either a local or a remote call. Excessive locking is avoided by using MPI RMA operations and memory windows. The model supported is single-writer, multiple-readers.

Concern 2: Support of highly concurrent metadata operations?

Upon opening a file, metadata are loaded from the PFS to the local RAM of the process that opened it. Then, Hermes randomly selects two other nodes and replicates metadata there. We do this to increase the availability of the metadata info and avoid saturation of one node's RAM. When another process wants to access the metadata, it randomly selects one of the replica copies and performs the get. If it needs to update the metadata, Hermes propagates the update to all replicas. This is synchronous to ensure consistency.

D. Hermes limitations: Hermes' DPE component implements our data placement policies based on the assumption that the user knows exactly what his/her workload involve, and thus, selecting the appropriate policy is not trivial. As a suggestion, the user can first profile his/her application using typical monitoring and profiling tools, such as Darshan [56], extract knowledge regarding the I/O behavior, and make the right policy choice.

4 Evaluation

4.1 Methodology

Overview: To evaluate Hermes, we have conducted two set of experiments. We first explored how Hermes' data placement policies handle different workloads and application characteristics using synthetic benchmarks. We then compare Hermes with state-of-the-art buffering platforms, namely Data Elevator and Cray's DataWarp, using real applications. As performance metric, we use the overall execution time in sec-

onds which we further divide to: i) time to write/read to/from buffers, and ii) time to flush buffers to PFS. Computation time is excluded since it is the same among all systems. As reference, we include a baseline of *no buffering* in which data are written/read directly to/from the PFS. We run all tests ten times and we report the average time.

Hardware: All experiments were conducted on Chameleon*. More specifically, we used the bare metal configuration with 32 client nodes (i.e., up to 1024 MPI ranks), 8 burst buffer nodes, and 16 PFS storage nodes. Each node has a dual Intel(R) Xeon(R) CPU E5-2670 v3 running at 2.30GHz with a total of 48 cores, and 128 GB RAM. Each burst buffer node is equipped with an SSD drive and each PFS node with an HDD. We emulated one NVMe device per client node by deploying a DRAM-based file system (i.e., RAMDISK) and imposing latency and bandwidth penalties to match the actual NVMe performance [52, 57, 58]. In order to correctly calculate the added latency and lowered bandwidth, we captured the performance characteristics of real NVMe devices present in the hierarchy appliances of Chameleon.

Table 1 lists all the hardware specifications and performance measurements and Figure 9 demonstrates the cluster topology resembling that of an typical HPC machine.

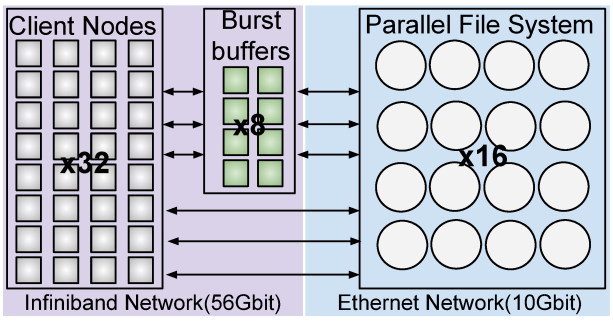

Fig.9. Cluster setup and topology.

Lastly, to better capture the architecture of a modern supercomputer, we setup our cluster topology as follows: all 32 client nodes and 8 burst buffers are interconnected with 56Gbps Infiniband network and the 16 storage nodes are connected to the rest via a 10Gbps Ethernet network.

Software: The operating system of the cluster is CentOS 7.1, the MPI version is Mpich 3.2, the PFS we used is OrangeFS 2.9.6, the in-memory key-value stores are Memcached 1.4.36 and Redis 4.0.6, and lastly the distributed queue we used is NATS Server 1.0.4.

Applications: We evaluate Hermes using our own synthetic benchmark that emulates common scientific application workloads such as alternation between computation - I/O phases, read -after-write, read-once, read-many etc. It uses POSIX-IO to issue requests to the file system and operates in a typical file-per-process pattern. We also use two real science applications: Vector Particle-In-Cell (VPIC) and Hardware Accelerated Cosmology Code (HACC). Both of these simulations perform computations and produce output files periodically that need to be persisted in PFS. Also, both demonstrate a periodic behavior with time steps (i.e., iterations) that include the checkpoint and restart as well as the analysis outputs produced by the simulations. We used 16 time steps for both simulations resulting to total I/O of 1TB.

4.2 Experimental Results

4.2.1 Synthetic Benchmarks

Our synthetic benchmark is highly tunable to generate workloads that can stress the buffering system under various use-cases. We designed two test-cases to evaluate Hermes' data placement policies.

Alternating Compute-I/O phases: In this test, each process first performs some computations (emu-

*https://www.chameleoncloud.org/about/chameleon/

Table 1. Testbed machine specifications

Device	RAM	NVMe	SSD	HDD
Model	M386A4G40DM0	Intel DC P3700	Intel DC S3610	ST9250610NS
Connection	DDR4 2133Mhz	PCIe Gen3 x8	SATA 6Gb/s	SATA 7200rpm
Capacity	128 GB(8GBx16)	1.2 TB	1.6 TB	2.4 TB
Latency	13.5 ns	20 us	55-66 us	4.16 ms
Max Read BW	13000 MB/s	2800 MB/s	550 MB/s	115 MB/s
Max Write BW	10000 MB/s	1900 MB/s	500 MB/s	95 MB/s
Test Config	32x client nodes	RamFS emulated	8x burst buffers	16x PFS servers
ReadBW tested	92647 MB/s	38674 MB/s	3326 MB/s	883 MB/s
WriteBW tested	86496 MB/s	33103 MB/s	2762 MB/s	735 MB/s

lated by *sleep*() calls) and then writes 64MB in a file-per-process fashion. We repeat this pattern 16 times with 1024 processes resulting in 1TB total I/O size. We vary the ratio of computation over I/O time to emulate three distinct types of applications: *data-intensive*, *compute-intensive*, and *balanced*. We assume that all data written to the buffers need to be also written to the disk-based remote PFS. Therefore, Hermes is configured in *persistent asynchronous* mode. We measure the overall time spent in I/O, in seconds, which consists of write-time and flush-time.

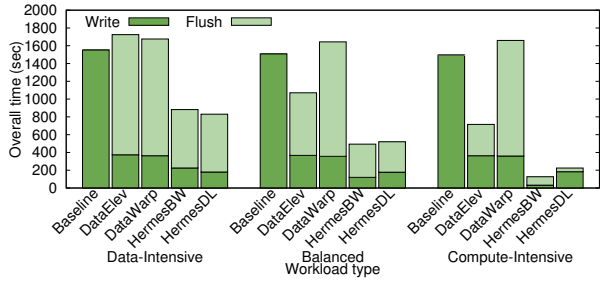

Fig.10. Benchmark: Alternating Compute-I/O phases.

Figure 10 shows the results. As it can be seen, the baseline writes directly to PFS (i.e., no flush-time) and maintains stable write performance regardless of the computation-I/O ratio. In Data Elevator and DataWarp, data are written to the burst buffers resulting to similar write-time between them. The difference in performance comes from data flushing. Data Elevator overlaps flushing with computation phases, and thus, as the computation-I/O ratio increases, flush-time decreases (i.e., flushing is hidden behind computation). On the other hand, DataWarp flushes data only once the application finishes and demonstrates stable flush-time regardless of the computation-I/O ratio. In Hermes, data are written in all layers of the DMSH (i.e., RAM, NVMe, and burst buffers in our system). We evaluate both *MaxBW* and *MaxLocality* data placement policies since they buffer data differently. MaxBW places data in a top-down fashion. It starts with RAM for the first iterations of the test, and once this layer is full, it first moves data down to NVMe to create space in RAM and then places the incoming iteration in RAM. On the other hand, MaxLocality uses layers concurrently. It writes the first iterations in RAM and once this layer is full it goes on to the next without any data movement between layers. It is clear that for data-intensive applications where the rate of incoming I/O is high, MaxBW's data movement between layers imposes some performance losses, and thus, MaxLocality's write performance is slightly higher. As the computation-I/O ratio increases however, MaxBW can overlap data movement between layers with computations. Therefore, for compute-intensive workloads, MaxBW outperforms MaxLocality by 4x in write-time since it ensures that incoming I/O can be written in RAM. For flushing, both policies leverage any computation time available to asynchronously flush buffer contents to PFS, similarly with

Data Elevator. However, Hermes flushes all layers of the DMSH concurrently which decreases flush-time significantly. In summary, in this test Hermes offers **8x** and **2x** higher write performance when compared to No Buffering baseline and state-of-the-art buffering platforms respectively.

Repetitive Read operations: In this test, the benchmark is configured to create a write-once, read-many workload. Each process first writes 32MB in a file-per-process approach and then reads back 32MB of data (not necessarily the same data). We have 16 phases of this pattern with 1024 processes aggregating the I/O to 1TB. We vary the repetition of read operations as follows: i) *Read-once*, where 32MB of data is read only once, ii) *Read-many x4*, where 8MB of data is read 4 times (i.e., still 32MB in total), and iii) *Read-many x16*, where 2MB of data is read 16 times. This pattern resembles workloads where portions of data such as metadata information, indices of files, etc., are frequently accessed creating a data hotness spectrum. In this test, we assume that buffers are used as scratch space (i.e., temporary I/O), and thus, Hermes is configured in *non-persistent* mode. The total time, in seconds, is divided into write-time and read-time.

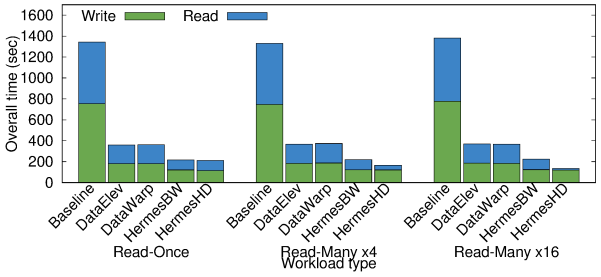

Fig.11. Benchmark: Repetitive Read operations.

As it can be seen in Figure 11, the baseline writes and reads directly from the PFS and maintains a stable performance irrespective of the workload type. In Data Elevator and DataWarp, data are written/read to/from the burst buffers respectively. This results to a considerable performance improvement over the baseline. Since repetitive read operations are treated as new, it shows stable performance across different workloads. In contrast, Hermes implements a *HotData* data placement policy to offer higher performance for this type of workloads. Since HotData will promote frequently accessed data in upper layers, repetitive read operations access data always from RAM resulting in significant performance boost for Read-many x4 and x16. On the other hand, MaxBW, while offering a competitive performance across the tested workloads, does not cache frequent accessed data in RAM and demonstrates a stable performance across the tested workloads. In summary, in this test Hermes offers **38x** and **11x** higher read performance when compared to No Buffering baseline and state-of-the-art buffering platforms respectively.

4.2.2 *Data-centric prefetching*

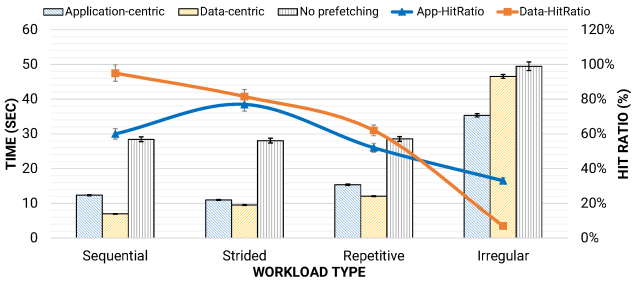

Fig.12. Application-centric vs. data-centric prefetching.

A system-wide prefetching approach has the advantage of observing how files are accessed across multiple processes or even applications. Hermes' prefetcher implements a data-centric logic where access statistics are collected and prefetching decisions are made based on how important a file block or region (i.e., segment in Hermes) is. In contrast, an application-centric prefetcher's main objective is to identify how each application accesses its data and make prefetching decisions accordingly. This approach might create scenarios where the cache can get polluted, or data are

fetched twice, or unnecessary evictions of prefetched data occur. To better understand the differences between an application-centric and data-centric prefetching approach and identify which workloads work best in each, we tested Hermes under the following scenario. We have 1024 processes in total organized in four different communicator groups representing different applications resembling a data analysis and visualization pipeline. Each process issues read requests on the same dataset. We tested four commonly-used patterns: sequential, strided, repetitive, and irregular access patterns. The prefetching cache size is configured to fit the total data size of two out of the four applications which means applications compete for access to this cache. For Hermes the prefetching cache is configured to fit one application's load in RAM and one in NVMe. Figure 12 demonstrates the evaluation results. As can be seen, for sequential, strided, and repetitive patterns, Hermes achieves **26%** higher performance when compared to an application-centric approach. Hermes is able to capture how data are accessed across applications or files and understand which segments are important to fetch from a global perspective. This results in zero cache evictions and no cache pollution. However, Hermes suffers from irregular patterns since created file heatmaps, that represent segment scoring, are uniformly flat (i.e., same heat throughout). Hence, the placement of segments in prefetching is effectively random which increases the miss ratio.

4.2.3 Real Applications

To test our system under real applications workload, we configured Hermes in *persistent asynchronous* mode since data need to be stored in the PFS for future access and selected the default data placement policy, *MaxBW*.

VPIC: Vector Particle-In-Cell (VPIC) is a general purpose simulation code for modeling kinetic plasmas in spatial multi-dimensions.This application demonstrates a write-only I/O access pattern where at the end of each time step, each process writes data to an HDF5 file. At the end of each step, VPIC writes a single HDF5 file containing properties of 8 million particles. VPIC tends to be extremely I/O intensive (i.e., write-only, write-heavy), since the portion of computation is small. During this evaluation we executed the application for 16 time steps. We strong scaled the application from 256 to 1024 total ranks and we measured the total time. In Figure 13 we report only the I/O time which consists of write-time (i.e., what the application experiences) and flush-time (i.e., persisting the data asynchronously). As it can be seen, all tested solutions scale linearly with the number of MPI ranks. In the largest tested scale of 1024 ranks, the baseline completed the test in 1192 seconds. Both Data Elevator and DataWarp wrote the entire dataset in 438 seconds. This is approximately a 2.5x improvement over the baseline. However, due to the higher bandwidth of the DMSH, Hermes' write performance is **5x** and **2x** higher than the baseline and the two buffering platforms we tested, respectively. When considering data flushing, Data Elevator overlaps small computations between each time step and flushes the contents of burst buffers in 1115 seconds whereas DataWarp flushes everything at the end in 1274 seconds. In contrast, Hermes leverages the computations but also the concurrency of the DMSH to flush all buffered data to PFS in 637 seconds. In summary, in this test, Hermes outperformed the baseline and state-of-the-art buffering platforms by **40%** and **85%** respectively.

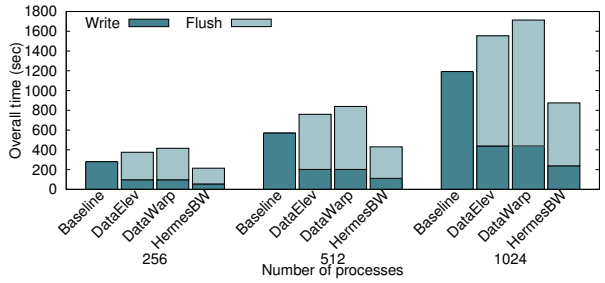

Fig.13. I/O Buffering performance with VPIC-IO.

HACC: Hardware Accelerated Cosmology Code (HACC) is a cosmological simulation that studies the formation of structure in collisionless fluids under the influence of gravity in an expanding universe. HACC has read-after-write workload where, at every step, simulation writes out a single shared file (i.e., MPI Collective I/O) that various analysis modules read back. This application demonstrates a read-after-write I/O access pattern where during each time step, each process reads back data previously written using MPI-Collective IO. During this evaluation we executed the application for 16 time steps. We strong scaled the application from 256 to 1024 total ranks and we measured the total time. In Figure 14 we report only the I/O time which consists of write-time, read-time, and flush-time. As it can be seen, all tested solutions scale linearly with the number of MPI ranks. In the largest tested scale of 1024 ranks, the baseline completed the test in 1313 seconds. Both Data Elevator and DataWarp performed I/O in 348 seconds. This is approximately a 3.7x improvement over the baseline. However, when considering data flushing, Data Elevator completed the test in 773 and DataWarp in 985 seconds effectively reducing the total improvement to 1.6x and 1.3x respectively. In contrast, Hermes completed the entire test in 494 seconds showcasing the potential of a DMSH system. The performance improvement is substantial when compared to No Buffering baseline with **7.5x** faster I/O operations. Hermes outperformed Data Elevator and DataWarp by **2x** due to higher bandwidth of the DMSH.

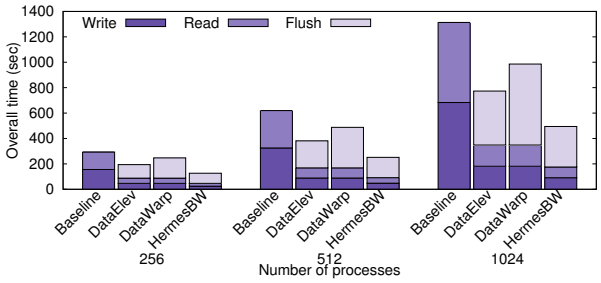

Fig.14. I/O Buffering performance with HACC-IO.

To evaluate the effectiveness of Hermes' data-centric prefetching approach, we performed scaling tests using two complex multi-phased scientific workflows namely Montage [59] and WRF*. We compare Hermes prefetching ability with Stacker [41] and KnowAc [21], one online and one offline prefetcher. Both of those solutions are configured to fetch data from burst buffers to the application's memory.

Montage This workflow is a collection of programs comprising an astronomical image mosaic engine based on MPI. It is a classical real world use-case of a workflow where multiple kernels share data for different purposes and access this common data concurrently. Each phase of building the mosaic takes an input from the previous phase and outputs intermediate data to the next one. Montage's workflow is highly read-intensive and iterative. Figure 15 shows the results for Montage. During this test, each process does 10 MB of I/O operations in 16 time steps for a total of 400 GB for the largest scale. We weak scaled the execution of Montage by increasing the number of processes from 320 to 2560. Required data are initially staged in the burst buffer nodes. The system is overall configured with prefetching cache organized in 1.5 GB RAM space, 2 GB in local NVMe drives and 400 GB burst buffer allocation. As can be seen, the best read performance is achieved by KnowAc,

*https://www.mmm.ucar.edu/weather-research-and-forecasting-model

a history-based prefetcher, since the prefetcher knows exactly what to load next. However, such approach suffers from prolonged profiling costs. Stacker avoids preprocessing steps and build its models as it goes, but demonstrated a lower hit ratio due to some cache conflicts and unwanted data evictions. Hermes is able to utilize all available tiers and performed the best, offering from **5-25%** better end-to-end performance when compared to Stacker and **10-30%** better than KnowAc (i.e., profile-cost plus run time). Note that all solutions scale nicely.

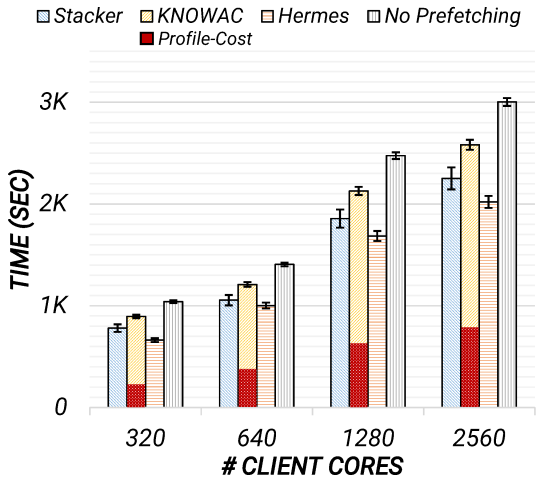

Fig.15. End-to-end performance of Montage scientific workflow.

WRF This workflow is a multi-phased mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting needs. WRF is multi-step iterative workflow where components of the simulation analyze observed and simulated data many times until the model converges. There are three distinct phases: pre-processing, main model, post-processing and visualization. Figure 16 shows the results for WRF. During this test, each process reads 8MB of data in 4 time steps for a total of 80GB across all scales (i.e., strong scale). Input data are assumed to be initially present in the burst buffer nodes. The system is configured with prefetching cache organized in 1.25 GB RAM space, 2 GB in local NVMe drives and 80 GB burst buffer allocation. Results confirm our previous observations with KnowAc having the best read time but additional profiling costs and Stacker demonstrating better end-to-end time over KnowAc. Hermes is able to utilize all tiers and scaled better than all solutions.

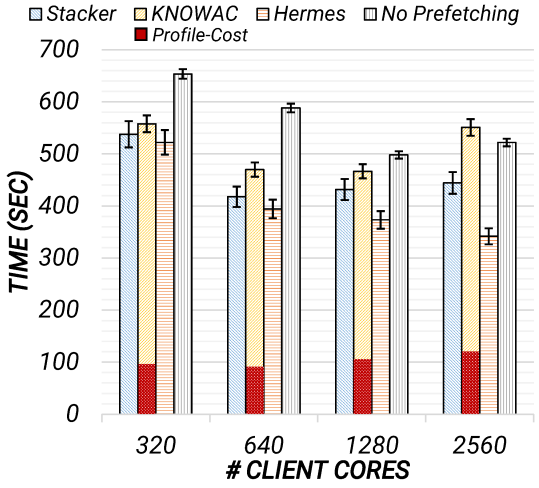

Fig.16. End-to-end performance of WRF scientific workflow.

5 Related Work

New hardware technologies have been developed and can be used to build new memory and storage hierarchies using non-volatile memory (NVRAM) such as phase-change memory (PCM) [58], memristors [60], and Flash memory [14]. Flash-based SSD technology has been widely studied [61], characterized [62], and evaluated for different application types [63, 64]. Researchers also advocate the use of shared buffer technologies, such as burst buffers [65], to accelerate I/O. Existing work has considered NVMe devices as a viable solution for I/O staging [15, 66]. Caulfield proposed Moneta [67], an architecture with NVRAM as an I/O device for HPC applications. Ekel extended Moneta with a real PCM device to understand the performance implications of using NVRAM [68]. Dong studied NVRAM for HPC application checkpointing [69]. Kannan studied NVRAM for I/O intensive benchmarks in Cloud envi-

ronments [15]. Wang proposed BurstMem [70], a technology for optimizing I/O using burst buffers. Sato et al., show how the burst buffers can boost performance of checkpointing tasks by 20x [71].

Active Buffers [72, 73] exploits one-sided communication for I/O processors to fetch data from compute processors' buffers and performs actual writing in the background while computation continues. IOLite [74], proposes a single shared memory per-node for leveraging inter-process communication and buffering of I/O. Such an approach led to 40% boost in performance. Nitzberg [75] proposes collective buffering algorithms for improving I/O performance by 100x on IBM SP2 at NASA Ames Research Center. PLFS [76] remaps an application's preferred data layout into one which is optimized for the underlying file system.

While all the above work emphasizes the benefits of using each technology individually, none introduced a complete I/O buffering platform that leverages the DMSH. The closest work to Hermes is Data Elevator [77] and its successor UniviStor [78], a new system that transparently moves data in a hierarchical system. The authors focused on systems equipped with burst buffers and demonstrated a 4x improvement over other state-of-the-art burst buffer management systems such as Cray's Datawarp*. However, they did not address local memory and local non-volatile devices such as NVMe. Hermes considers both local resources and shared resources like burst buffers. Furthermore, Hermes extends buffering into remote resources and tackles data movement to a more complicated landscape of I/O-capable devices.

In addition to that relevant research, we identify the following work for data prefetching. Diskseen [38], tracks the locations and access times of disk blocks. Based on analysis of their temporal and spatial relationships, it seeks to improve the sequentiality of disk accesses and overall prefetching performance. However, disk blocks do not carry file semantics and relationships between segments. During Hermes design and development, we drew partial motivation from a cache replacement algorithm presented in [88] where frequency and recency of a memory page can both influence the eviction of the page. Hermes' segment scoring resembles in a sense a similar approach where we target segment with score based on access frequency and recency.

6 Conclusions

To increase I/O performance, modern storage systems are presented in a new memory and storage hierarchy, called Deep Memory and Storage Hierarchy. However, data movement among the layers is significantly complex, making it harder to take advantage of the high-speed and low-latency storage systems. Additionally, each layer of the DMSH is an independent system that requires expertise to manage, and the lack of automated data movement between tiers is a significant burden currently left to the users.

In this paper, we present the design and implementation of Hermes: a new, heterogeneous-aware, multi-tiered, dynamic, and distributed I/O buffering system. Hermes enables, manages, and supervises I/O buffering into the DMSH and offers a buffering platform that can be application- and system-aware, and thus, hide lower level details allowing the user to focus on his/her algorithms. Hermes aims to maximizing productivity, increasing resource utilization, abstracting data movement, maximizing performance, and supporting a wide range of scientific applications and domains. We have presented three novel data placement policies to efficiently utilize all layers of the new memory and storage hierarchy as well as three novel techniques to perform

*http://www.cray.com/sites/default/files/resources/ CrayXC40-DataWarp.pdf

memory, *metadata*, and *communication* management in hierarchical buffering systems. Our evaluation results prove Hermes' sound design and show a 8x improvement compared to systems without I/O buffering support. Additionally, Hermes outperforms by more than 2x state-of-the-art buffering platforms such as Data Elevator and Cray's Datawarp.

References

[1] Kitchin, Rob. "Big Data, new epistemologies and paradigm shifts." Big data & society 1, no. 1 (2014): 2053951714528481.

[2] Reinsel, David, John Gantz, and John Rydning. "Data age 2025: The evolution of data to life-critical." Don't Focus on Big Data (2017). White paper

[3] Tansley, Stewart, and Kristin M. Tolle. The fourth paradigm: data-intensive scientific discovery. Edited by Anthony JG Hey. Vol. 1. Redmond, WA: Microsoft research, 2009.

[4] Thakur, Rajeev, William Gropp, and Ewing Lusk. "Data sieving and collective I/O in ROMIO." In Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation, pp. 182-189. IEEE, 1999.

[5] Folk, Mike, Albert Cheng, and Kim Yates. "HDF5: A file format and I/O library for high performance computing applications." In Proceedings of Supercomputing, vol. 99, pp. 5-33. 1999.

[6] Braam, Peter. "The Lustre storage architecture." arXiv preprint arXiv:1903.01955 (2019).

[7] Schmuck, Frank B., and Roger L. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In FAST, vol. 2, no. 19. 2002.

[8] Ross, Robert B., and Rajeev Thakur. "PVFS: A parallel file system for Linux clusters." In Proceedings of the 4th annual Linux showcase and conference, pp. 391-430. 2000.

[9] Khaleel, Mohammad A. Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale - February 2-4, 2010, Washington, D.C., report, February 6, 2011; Richland, Washington. (https://digital.library.unt.edu/ark:/67531/metadc841613/: accessed August 14, 2019), University of North Texas Libraries, Digital Library, https://digital.library.unt.edu; crediting UNT Libraries Government Documents Department.

[10] Dongarra, Jack, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai et al. "The international exascale software project roadmap." International Journal of High Performance Computing Applications 25, no. 1 (2011): 3-60.

[11] Reed, Daniel A., and Jack Dongarra. "Exascale computing and big data." Communications of the ACM 58, no. 7 (2015): 56-68.

[12] Shalf, John, Sudip Dosanjh, and John Morrison. "Exascale computing technology challenges." In International Conference on High Performance Computing for Computational Science, pp. 1-25. Springer, Berlin, Heidelberg, 2010.

[13] Bent, John, Gary Grider, Brett Kettering, Adam Manzanares, Meghan McClelland, Aaron Torres, and Alfred Torrez. "Storage challenges at los alamos national lab." In 012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1-5. IEEE, 2012.

[14] Caulfield, Adrian M., Laura M. Grupp, and Steven Swanson. "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications." ACM Sigplan Notices 44, no. 3 (2009): 217-228.

[15] Kannan, Sudarsun, Ada Gavrilovska, Karsten Schwan, Dejan Milojicic, and Vanish Talwar. "Using active NVRAM for I/O staging." In Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities, pp. 15-22. ACM, 2011.

[16] Caulfield, Adrian M., Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing." In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11. IEEE Computer Society, 2010.

[17] Lockwood, Glenn K., Damian Hazen, Quincey Koziol, R. S. Canon, Katie Antypas, Jan Balewski, Nicholas Balthaser et al. "Storage 2020: A Vision for the Future of HPC Storage." (2017). Technical Report. NERSC.

[18] Li, Jianwei, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. "Parallel netCDF: A high-performance scientific I/O interface." In SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, pp. 39-39. IEEE, 2003.

[19] Lofstead, Jay F., Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. "Flexible io and integration for scientific codes through the adaptable io system (adios)."

In Proceedings of the 6th international workshop on Challenges of large applications in distributed environments, pp. 15-24. ACM, 2008.

[20] Chang, Fay, and Garth A. Gibson. "Automatic I/O hint generation through speculative execution." In Proceedings of the third symposium on Operating systems design and implementation, pp. 1-14. USENIX Association, 1999.

[21] He, Jun, Xian-He Sun, and Rajeev Thakur. "Knowac: I/O prefetch via accumulated knowledge." In 2012 IEEE International Conference on Cluster Computing, pp. 429-437. IEEE, 2012.

[22] Dong, Bin, Teng Wang, Houjun Tang, Quincey Koziol, Kesheng Wu, and Suren Byna. "ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage." In 2018 IEEE International Conference on Big Data (Big Data), pp. 211-220. IEEE, 2018.

[23] Buyya, Rajkumar, Rodrigo N. Calheiros, and Amir Vahid Dastjerdi, eds. Big data: principles and paradigms. Morgan Kaufmann, 2016.

[24] Kune, Raghavendra, Pramod Kumar Konugurthi, Arun Agarwal, Raghavendra Rao Chillarige, and Rajkumar Buyya. "The anatomy of big data computing." Software: Practice and Experience 46, no. 1 (2016): 79-105.

[25] Kougkas, Anthony, Hariharan Devarajan, Xian-He Sun, and Jay Lofstead. "Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Buffers." In 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 290-301. IEEE, 2018.

[26] Xie, Bing, Yezhou Huang, Jeffrey S. Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. "Predicting output performance of a petascale supercomputer." In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pp. 181-192. ACM, 2017.

[27] Kim, Youngjae, Raghul Gunasekaran, Galen M. Shipman, David A. Dillow, Zhe Zhang, and Bradley W. Settlemyer. "Workload characterization of a leadership class storage cluster." In 2010 5th Petascale Data Storage Workshop (PDSW'10), pp. 1-5. IEEE, 2010.

[28] Mi, Ningfang, Alma Riska, Qi Zhang, Evgenia Smirni, and Erik Riedel. "Efficient management of idleness in storage systems." ACM Transactions on Storage (TOS) 5, no. 2 (2009): 4.

[29] Ahern, Sean, Sadaf R. Alam, Mark R. Fahey, Rebecca J. Hartman-Baker, Richard F. Barrett, Ricky A. Kendall, Douglas B. Kothe et al. Scientific application requirements for leadership computing at the exascale. No. ORNL/TM-2011/250. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States); Center for Computational Sciences, 2007.

[30] Carns, Philip, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. "Understanding and improving computational science storage access through continuous characterization." ACM Transactions on Storage (TOS) 7, no. 3 (2011): 8.

[31] Dundas, James, and Trevor Mudge. "Improving data cache performance by pre-executing instructions under a cache miss." In International Conference on Supercomputing, pp. 68-75. 1997.

[32] Doweck J. 2006. Shared memory access. White paper, Intel Research Website. http://download.intel.com/technology/architecture/sma.pdf

[33] Mutlu, Onur, Jared Stark, Chris Wilkerson, and Yale N. Patt. "Runahead execution: An alternative to very large instruction windows for out-of-order processors." In The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings., pp. 129-140. IEEE, 2003.

[34] Qadri, Muhammad Yasir, Nadia N. Qadri, Martin Fleury, and Klaus D. McDonald-Maier. "Energy-efficient data prefetch buffering for low-end embedded processors." Microelectronics Journal 62 (2017): 57-64.

[35] Sun, Xian-He, Surendra Byna, and Yong Chen. "Server-based data push architecture for multi-processor environments." Journal of Computer Science and Technology 22, no. 5 (2007): 641-652.

[36] Zhou, Huiyang. "Dual-core execution: Building a highly scalable single-thread instruction window." In 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05), pp. 231-242. IEEE, 2005.

[37] Cao, Pei, Edward W. Felten, Anna R. Karlin, and Kai Li. "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling." ACM Transactions on Computer Systems (TOCS) 14, no. 4 (1996): 311-343.

[38] Ding, Xiaoning, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch." In USENIX Annual Technical Conference, vol. 7, pp. 261-274. 2007.

[39] Alexander C Klaiber and Henry M Levy. 1991. An architecture for software-controlled data prefetching. In ACM SIGARCH Computer Architecture News, Vol. 19. ACM, USA, 43-53.

[40] Mowry, Todd, and Anoop Gupta. "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors." Journal of parallel and Distributed Computing 12, no. 2 (1991): 87-106.

[41] Subedi, Pradeep, Philip Davis, Shaohua Duan, Scott Klasky, Hemanth Kolla, and Manish Parashar. "Stacker: an autonomic data movement engine for extreme-scale data staging-based in-situ workflows." In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, p. 73. IEEE Press, 2018.

[42] Cherubini, Giovanni, Yusik Kim, Mark Lantz, and Vinodh Venkatesan. "Data prefetching for large tiered storage systems." In 2017 IEEE International Conference on Data Mining (ICDM), pp. 823-828. IEEE, 2017.

[43] Joo, Yongsoo, Sangsoo Park, and Hyokyung Bahn. "Exploiting i/o reordering and i/o interleaving to improve application launch performance." ACM Transactions on Storage (TOS) 13, no. 1 (2017): 8.

[44] Abbasi, Hasan, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. "Datastager: scalable data staging services for petascale applications." Cluster Computing 13, no. 3 (2010): 277-290.

[45] Bengio, Yoshua. "Markovian models for sequential data." Neural computing surveys 2, no. 199 (1999): 129-162.

[46] V Thilaganga, M Karthika, and M Maha Lakshmi. 2017. A Prefetching Technique Using HMM Forwardand Backward Chaining for the DFS in Cloud. Asian Journal of Computer Science and Technology 6, 2 (2017), 23-26.

[47] Tran, Nancy, and Daniel A. Reed. "Automatic ARIMA time series modeling for adaptive I/O prefetching." IEEE Transactions on parallel and distributed systems 15, no. 4 (2004): 362-377.

[48] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2014. Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction. In SC'14:Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, USA, 623-634.

[49] Yifeng Luo, Jia Shi, and Shuigeng Zhou. 2017. JeCache: just-enough data caching with just-in-time prefetching for big data applications. In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, USA, 2405-2410.

[50] Daniel, Gwendal, Gerson Sunye, and Jordi Cabot. "Prefetchml: a framework for prefetching and caching models." In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 318-328. ACM, 2016.

[51] Xu, Rui, Xi Jin, Linfeng Tao, Shuaizhi Guo, Zikun Xiang, and Teng Tian. "An efficient resource-optimized learning prefetcher for solid state drives." In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 273-276. IEEE, 2018.

[52] Wu, Kai, Yingchao Huang, and Dong Li. "Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory." In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 58. ACM, 2017.

[53] Snyder, Bruce, Dejan Bosanac, and Rob Davies. "Introduction to apache activemq." Active MQ in Action (2017): 6-16.

[54] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." In Proceedings of the NetDB, pp. 1-7. 2011.

[55] Zawislak, Dawid, Brian Toonen, William Allcock, Silvio Rizzi, Joseph Insley, Venkatram Vishwanath, and Michael E. Papka. "Early investigations into using a remote ram pool with the vl3 visualization framework." In 2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), pp. 23-28. IEEE, 2016.

[56] Carns, Philip, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. "24/7 characterization of petascale I/O workloads." In 2009 IEEE International Conference on Cluster Computing and Workshops, pp. 1-10. IEEE, 2009.

[57] Dulloor, Subramanya R., Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. "System software for persistent memory." In Proceedings of the Ninth European Conference on Computer Systems, p. 15. ACM, 2014.

[58] Moinuddin K Qreshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. ACM SIGARCH Computer Architecture News 37, 3 (2009), 24-33.

[59] GB Berriman, JC Good, AC Laity, and M Kong. 2008. The Montage image mosaic service: custom image mosaics on-demand. Astronomical Data Analysis Software and Systems ASP Conference Series 394, 2 (2008), 83-102.

[60] Strukov, Dmitri B., Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. "The missing memristor found." nature 453, no. 7191 (2008): 80.

[61] Joo, Yongsoo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. "FAST: Quick Application Launch on Solid-State Drives." In FAST, pp. 259-272. 2011.

[62] El Maghraoui, Kaoutar, Gokul Kandiraju, Joefon Jann, and Pratap Pattnaik. "Modeling and simulating flash based solid-state disks for operating systems." In Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering, pp. 15-26. ACM, 2010.

[63] Andersen, David G., Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. "FAWN: A fast array of wimpy nodes." In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 1-14. ACM, 2009.

[64] Chen, Shimin. "FlashLogging: exploiting flash devices for synchronous logging performance." In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 73-86. ACM, 2009.

[65] Bhimji, Wahid, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson et al. Accelerating Science with the NERSC Burst Buffer Early User Program. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2016.

[66] Kang, Sooyong, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. "Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices." IEEE Transactions on Computers 58, no. 6 (2008): 744-758.

[67] Caulfield, Adrian M., Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories." In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 385-395. IEEE Computer Society, 2010.

[68] Akel, Ameen, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. "Onyx: A Prototype Phase Change Memory Storage Array." HotStorage 1 (2011): 1.

[69] Dong, Xiangyu, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems." In Proceedings of the conference on high performance computing networking, storage and analysis, p. 57. ACM, 2009.

[70] Wang, Teng, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. "Burstmem: A high-performance burst buffer system for scientific applications." In 2014 IEEE International Conference on Big Data (Big Data), pp. 71-79. IEEE, 2014.

[71] Sato, Kento, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. De Supinski, Naoya Maruyama, and Satoshi Matsuoka. "A user-level infiniband-based file system and checkpoint strategy for burst buffers." In 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 21-30. IEEE, 2014.

[72] Ma, Xiaosong, Marianne Winslett, Jonghyun Lee, and Shengke Yu. "Faster collective output through active buffering." In Proceedings 16th International Parallel and Distributed Processing Symposium, pp. 8-pp. IEEE, 2001.

[73] Ma, Xiaosong, Marianne Winslett, Jonghyun Lee, and Shengke Yu. "Improving MPI-IO output performance with active buffering plus threads." In Proceedings International Parallel and Distributed Processing Symposium, pp. 10-pp. IEEE, 2003.

[74] Pai, Vivek S., Peter Druschel, and Willy Zwaenepoel. "IO-Lite: A unified I/O buffering and caching system." In OSDI, vol. 99, pp. 15-28. 1999.

[75] Nitzberg, Bill, and Virginia Lo. "Collective buffering: Improving parallel I/O performance." In Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183), pp. 148-157. IEEE, 1997.

[76] Bent, John, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. "PLFS: a checkpoint filesystem for parallel applications." In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 21. ACM, 2009.

[77] Dong, Bin, Suren Byna, Kesheng Wu, Hans Johansen, Jeffrey N. Johnson, and Noel Keen. "Data elevator: Low-contention data movement in hierarchical storage system." In 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pp. 152-161. IEEE, 2016.

[78] Wang, Teng, Suren Byna, Bin Dong, and Houjun Tang. "UniviStor: Integrated Hierarchical and Distributed Storage for HPC." In 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 134-144. IEEE, 2018.

[79] Lee, Donghee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies." IEEE transactions on Computers 12 (2001): 1352-1361.

Appendix

A. Maximum Application Bandwidth (MaxBW):

$$\boldsymbol{DPE_{MaxBW}(s, C_i)} = \begin{Bmatrix} (s/BW_i) * A_i & , s \leq C_i \\ \min \begin{pmatrix} DPE(s, C_{i+1}) \\ DPE(C_i, C_i) + DPE(s - C_i, C_{i+1}) \\ Move(s - C_i, i+1) + DPE(s, C_i)) \end{pmatrix} & , s > C_i \end{Bmatrix} \quad (2)$$

where s is the request size, C is a layer's remaining capacity in MBs, i is the current layer, BW is the bandwidth in MB/s, A is the access latency in ms, and $Move(min\_size, dest)$ triggers data organizer to recursively move at least min_size data to $dest$ layer.

B. Maximum Data Locality:

$$\boldsymbol{DPE_{MaxLocality}(s, d, L_i, R_i)} =$$

$$\begin{Bmatrix} (s/BW_i) * d & , L_i \ \& \ s \leq R_i \\ \min \begin{pmatrix} (s/BW_i) * (d+1) \\ DPE(s, d, L_{i+1}, R_{i+1}) \end{pmatrix} & , !L_i \ \& \ s \leq R_i \\ \min \begin{pmatrix} DPE(s, d, L_{i+1}, R_{i+1}) \\ DPE(R_i, d, L_i, R_i) + DPE(s - R_i, d, L_{i+1}, R_{i+1}) \\ ReOrganize(s - R_i) + DPE(s, d, L_i, R_i)) \end{pmatrix} & , s > R_i \end{Bmatrix} \quad (3)$$

where s is the request size, d is the degree of data dispersion into DMSH, L is the locality of a dispersion unit in layer (i.e., if it exists in this layer or not), R is a layer's capacity threshold, i is the current layer, BW is the bandwidth in MB/s, and $ReOrganize(min\_size)$ is a function that triggers data organizer to recursively move at least min_size data to maintain the locality of a dispersion unit.

C. Hot-data:

$$\boldsymbol{DPE_{HotData}(s, h, H_i, C_i)} =$$

$$\begin{Bmatrix} (s/C_i)/BW & , h \geq H_i \ \& \ s \leq C_i \\ \min \begin{pmatrix} DPE(s, h-1, H_{i+1}, C_{i+1}) \\ DPE(C_i, h, H_i, C_i) + DPE(s - C_i, h-1, H_{i+1}, C_{i+1}) \\ Evict(s - C_i, h, i+1) + DPE(s, h, H_i, C_i)) \end{pmatrix} & , h \geq H_i \ \& \ s > C_i \\ \min \begin{pmatrix} DPE(s, h+1, H_i, C_i) \\ DPE(s, h, H_{i+1}, C_{i+1}) \end{pmatrix} & , h < H_i \ \& \ s \leq C_i \\ \min \begin{pmatrix} DPE(C_i, h+1, H_i, C_i) + DPE(s - C_i, h, H_{i+1}, C_{i+1}) \\ DPE(s, h, H_{i+1}, C_{i+1}) \end{pmatrix} & , h < H_i \ \& \ s > C_i \end{Bmatrix} \quad (4)$$

where s is the request size, h is the file's hotness score, H is the minimum hotness score present in a layer, C is a layer's remaining capacity in MBs, i is the current layer, BW is the bandwidth in MB/s, and $Evict(min\_size, score, dest)$ is a function that triggers data organizer to recursively move at least min_size data to the $dest$ layer with $score$ hotness.