Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Bu ering System

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

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 perfor- mance. New hardware technologies, such as NVMe and SSD, have been introduced in burst bu er installations to reduce the pressure for external storage and boost the burstiness of modern I/O systems. e 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 signi cantly more complex even without considering heterogeneity. How to e ciently utilize the DMSH is a subject of research facing the HPC community. In this paper, we present the design and implementa- tion of Hermes: a new, heterogeneous-aware, multi-tiered, dynamic, and distributed I/O bu ering system. Hermes enables, manages, su- pervises, and, in some sense, extends I/O bu ering to fully integrate into the DMSH. We introduce three novel data placement policies to e ciently utilize all layers and we present three novel techniques to perform memory, metadata, and communication management in hierarchical bu ering systems. Our evaluation shows that, in ad- dition to automatic data movement through the hierarchy, Hermes can signi cantly accelerate I/O and outperforms by more than 2x state-of-the-art bu ering platforms.

CCS CONCEPTS

•Information systems→ Distributed storage; Record and bu er management; Main memory engines; Storage class memory; Cloud based storage; Hierarchical storage management; •Hardware → External storage; Emerging architectures; Memory and dense storage;

KEYWORDS I/O bu ering, Heterogeneous bu ering, Layered bu ering, Deep memory hierarchy, Burst bu ers

ACM Reference format: Anthony Kougkas, Hariharan Devarajan, Xian-He Sun. 2018. Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Bu ering System. In Proceedings of HPDC ’18: International Symposium on High-Performance Parallel and Distributed Computing, Tempe, AZ, USA, June 11–15, 2018 (HPDC ’18), 12 pages. DOI: 10.1145/3208040.3208059

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro t or commercial advantage and that copies bear this notice and the full citation on the rst page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permi ed. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior speci c permission and/or a fee. Request permissions from permissions@acm.org. HPDC ’18, Tempe, AZ, USA © 2018 ACM. 978-1-4503-5785-2/18/06. . .$15.00 DOI: 10.1145/3208040.3208059

1 INTRODUCTION Data-driven science is a reality and in fact, is now driving scien- ti c discovery [28]. An International Data Corp. (IDC) report [44] predicts that by 2025, the global data volume will grow to 163 ze abytes, ten times the 16.1ZB of data generated in 2016. e evo- lution 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 scienti c instruments and sensor networks collect extreme amounts of data and push for more capable storage sys- tems [23]. Modern I/O systems have been developed and highly optimized through the years. Popular interfaces and standards such as POSIX I/O, MPI-IO [51], and HDF5 [22] expose data to the appli- cations and allow users to interact with the underlying le system through extensive APIs. In a large scale environment, the underly- ing le system is usually a parallel le system (PFS) with Lustre [41], GPFS [47], PVFS2 [45] being some popular examples. However, as we move towards the exascale era, most of these storage systems face signi cant challenges in performance, scalability, complexity, and limited metadata services [7, 19], creating the so called I/O bo leneck which will lead to less scienti c productivity [43, 48].

To reduce the I/O performance gap, modern storage subsystems are going through extensive changes, by adding additional lev- els of memory and storage in a hierarchy [5]. Newly emerging hardware technologies such as High-Bandwidth Memory (HBM), Non-Volatile RAM (NVRAM), Solid-State Drives (SSD), and ded- icated bu ering nodes (e.g., burst bu ers) 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) [12, 26]. For example, Cori system at the National Energy Research Scienti c Computing Center (NERSC) [38], uses CRAY’s Datawarp technology [16]. Los Alamos National Laboratory Trin- ity supercomputer [34] uses burst bu ers 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 bu ering [54].

As multiple layers of storage are added into HPC systems, the complexity of data movement among the layers increases signi - cantly, making it harder to take advantage of the high-speed and low-latency storage systems [10]. 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 signi cant burden currently le to the users [32]. Furthermore, popular I/O middleware, such as HDF5, PnetCDF [31], and ADIOS [33], are con gured to operating with the traditional memory-to-disk I/O endpoints. is middleware provides great value by isolating users

HPDC ’18, June 11–15, 2018, Tempe, AZ, USA Anthony Kougkas, Hariharan Devarajan, Xian-He Sun

from the complex e ort to extract peak performance from the un- derlying storage system, but it will need to be updated to handle the transition to a multi-tiered I/O con guration [32]. ere is a need to seamlessly and transparently support access to DMSH.

In this paper, we present the design and implementation of Her- mes: a new, heterogeneous-aware, multi-tiered, dynamic, and dis- tributed I/O bu ering system. Hermes enables, manages, and super- vises I/O bu ering into DMSH and o ers: a) vertical and horizontal distributed bu ering in DMSH (i.e., access data to/from di erent levels locally and across remote nodes), b) selective layered data placement (i.e., bu er data partially or entirely in various levels of the hierarchy), c) dynamic bu ering via system pro ling (i.e., change the bu ering schema dynamically by monitoring the system sta- tus such as capacity of bu ers, messaging tra c, etc.). Hermes accelerates applications’ I/O access by transparently bu ering data in DMSH. Data can be moved through the hierarchy e ortlessly and therefore, applications have a capable, scalable, and reliable middleware so ware to navigate the I/O challenges towards the exascale era. Lastly, by supporting both POSIX and HDF5 interfaces, Hermes o ers ease-of-use to a wide-range of scienti c applications.

e contributions of this work include:

• presenting the design and implementation of Hermes: a new, heterogeneous-aware, multi-tiered, dynamic, and dis- tributed I/O bu ering system (Section 3.1).

• introducing three novel data placement policies to e - ciently 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 commu- nication management in hierarchical bu ering systems (Section 3.3.2).

• evaluating Hermes’ design and technical innovations show- ing that our solution can grant be er performance com- pared to the state-of-the-art bu ering 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 require- ments and complexity [15]. Many applications spend signi cant time of the overall execution in performing I/O making storage a vital component in performance [56]. Furthermore, scienti c appli- cations o en demonstrate bursty I/O behavior [27, 37]. Typically, in HPC workloads, short, intensive, phases of I/O activities, such as checkpointing and restart, periodically occur between longer computation phases [1, 8]. e intense and periodic nature of I/O operations stresses the underlying parallel le system and thus, stalls the application. To appreciate how important and challenging the I/O performance of a system is, one needs to deeply under- stand the I/O behavior of modern scienti c applications. More and more scienti c applications generate very large datasets, and the development of several disciplines greatly relies on the analysis of massive data. We highlight some scienti c domains that are increasingly relying on High-Performance Data Analytics (HPDA), the new generation of data-intensive applications, which involve su cient data volumes and algorithmic complexity to require HPC

resources: Computational Biology: e National Center for Biotech- nology Innovation maintains the GenBank database of nucleotide sequences, which doubles in size every 10 months. e 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 tele- scope in the world which produces staggering data as presented in the keynote speech during the 2017 SC conference. As high- lighted, the incoming images are of 10 PBs and the produced 3D image is 1 PB each. High-Energy Physics: e 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 bu ers be- tween the main memory and the disks are of particular relevance in mitigating the periodic burstiness of I/O. e new DMSH promises to o er a solution that can e ciently support scienti c discov- ery 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 stor- age systems is a cost-e ective strategy to reduce the I/O latency of HPC applications. However, while DMSH systems o er higher I/O performance, data movement between the layers of the hierarchy is complex and signi cantly challenging to manage. Moreover, there is no so ware 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 les while encapsulating general purpose optimizations. However, the actual optimization strategy of these middleware layers is dependent on the underlying le system so ware 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 le to the user. Ideally, the presence of multiple layers of storage should be transparent to applications without having to sacri ce performance or increase programming di culty. System so ware and a new middleware so- lution 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 so ware stack, regardless of the underlying storage. Deep memory and storage hierarchies require a scalable, reliable, and high-performance so ware to e ciently and transparently manage data movement. New data placement and ushing policies, memory and metadata management, and an e cient I/O communi- cation fabric is required to address DMSH complexity and realize its potential. We believe that a radical departure from the existing so ware stack for the scienti c communities is not realistic. ere- fore, 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

Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Bu ering System HPDC ’18, June 11–15, 2018, Tempe, AZ, USA

Applications High Level I/O Libraries (pNetCDF,HDF5, etc.,)

POSIX

MPI-IO

Compute nodes Node 1 Application ...

Node n Application

**External Services** 1. Application Orchestrator

**Hermes Library**

API

2. System Profiler **Hermes**

**Hermes**3. User-defined Schema Parser

Data Placement Engine

**RAM**

**RAM**

**Deep Memory and Storage Hierarchy** 1. Local RAM **NVMe**

**NVMe**

2. Remote RAM 3. Local NVMe

Cache Manager

**DMS**Burst Buffer Nodes Prefetcher

Metadata Manager Messaging service

Data Organizer

4. Remote NVMe 5. Burst Buffers

RAM NVMe I/O Clients Burst

File Buffers

System

6. Parallel File System

RAM

**DMSH Hardware**

**H**

**Data Movement:** 1. Vertical local NVMe

2. Vertical remote SSD (i.e.,Burst Buffers)

3. Horizontal HDD (i.e., Parallel File System)

Parallel File System(PFS)

Figure 1: So ware stack and Hermes internal design.

Figure 2: Hermes internal design.

supports existing widely popular I/O libraries such as MPI-IO and

be present and positioned close to the compute nodes. Finally, a HDF5 which makes our solution highly exible and production-

remote PFS supports all compute nodes with persistence and fault ready. We envision a bu ering platform that can be application- and

tolerance as important features. Hermes is a platform that aims to system-aware, and thus, hide lower level details allowing the user

enable e cient access to the layers of DMSH and as such we distin- to focus on his/her algorithms. We strive for maximizing produc-

guish two data paths: a vertical and a horizontal hierarchy. Vertical tivity, increasing resource utilization, abstracting data movement,

hierarchy refers to data movement within a compute node and all maximizing performance, and supporting a wide range of scienti c

the way down to the burst bu ers and PFS. Horizontal hierarchy applications and domains.

refers to sending data to another compute node’s RAM or NVMe 3 DESIGN AND IMPLEMENTATION 3.1 Hermes Architecture

3.1.1 Design overview. Hermes is designed as a middleware layer - si ing between applications and DMSH as shown in Fig- ure 1. As a middleware library, Hermes captures I/O calls, both

device. e horizontal data movement is greatly optimized if there is an RDMA-capable network but Hermes can also support systems with no RDMA. erefore, a DMSH system could consist of several layers, performance-wise, such as local RAM, remote RAM, local NVMe, remote NVMe, burst bu ers, and PFS (numbered in g. 2).

POSIX and HDF5 (i.e., fopen, fread, fwrite, and H5Fcreate, H5Dread

3.1.2 Internal components. Figure 1 demonstrates the design of etc.) and redirects them to di erent layers of DMSH. Legacy ap-

Hermes library and all the internal components that work together plications can easily connect to Hermes by simple linking (i.e.,

to achieve an e cient, transparent, and easy-to-use data access in LD PRELOAD) or recompiling the code with our library. ere

all layers of a DMSH (i.e., both vertically and horizontally). e are no changes to user code and there is no need to upgrade to a

main Hermes library is complemented by a set of tools and services di erent work ow. We design Hermes to easily work with existing

that help achieve broader goals such as multi-tenancy, adaptability, so ware. Our goal is to maximize user productivity by making

etc. Brief description of each component’s responsibilities: I/O bu ering transparent. Furthermore, Hermes also provides a

API: e API is responsible to intercept all I/O calls from the ap- new bu ering API for users who want to explicitly take control

plications. It also calculates the operations to be carried out by the of the data movement between layers of DMSH. is mode also

bu ering nodes in case of an active bu ering scenario. allows Hermes to perform active bu ering where data is shipped

Data Placement Engine: is engine is responsible to map data to the bu er nodes along with speci c instructions or operations

onto DMSH. In other words, the data placement engine calculates to be performed on them. For example, a user can pass a set of

the data destination, where in the hierarchy should the data be redi- integers to Hermes instructing it to rst store them to the bu er

rected. It maps data according to various data placement policies. nodes, then sort them, compress the sorted list and lastly persist the

Data Organizer: e main responsibility of this component is to nal result to the remote PFS. is ow can be easily executed by a

move data between the layers of DMSH. It is triggered by other com- series of hinting mechanisms (i.e., ags) that Hermes provides to

ponents according to certain criteria which makes it an event-based the user. Our hinting mechanism is a simple bit encryption which

component. For instance, if there is no space le in NVMe, data indicates predetermined operations like sorting, compression/de-

organizer is triggered to move data down to the burst bu ers and compression, deduplication and others. For user de ned operations,

thus freeing space in NVMe. is component is responsible to carry Hermes provides a bootstrapping mechanism in which the user can

out all data movement either for prefetching reasons, evictions, lack submit his/her functions. e library will then compile and place

of space, or hotness of data etc. the executables to a registry of operations to be handled by the

Metadata Manager: e MDM maintains two types of metadata bu ering nodes. Reserved bits are used for user-de ned operations.

information: user’s and Hermes library’s internal metadata. Since e high-level architecture of Hermes can be seen in Figure 2. In

Hermes can transparently bu er data by intercepting I/O calls, DMSH systems, besides the main memory, every compute node

MDM keeps track of user’s metadata operations (i.e., les, directo- might be equipped with an NVMe device or even an SSD. Addition-

ries, permissions etc.) while consulting the underlying PFS. Addi- ally, shared bu ering nodes, such as burst bu ers, will most likely

tionally, since data can be bu ered anywhere in the hierarchy, MDM

HPDC ’18, June 11–15, 2018, Tempe, AZ, USA Anthony Kougkas, Hariharan Devarajan, Xian-He Sun

tracks the locations of all bu ered data and internal temporary les that contain user les. Cache manager: is component is responsible to handle all bu ers inside Hermes. It is equipped with several cache replace- ment policies such as least recently used (LRU) and least frequently used (LFU). It works in conjunction with the prefetcher. It can be con gured to hold ”hot” data for be er I/O latency. It is also responsible to implement application-aware caching schemas. Prefetcher: is component is performance-driven. It implements several typical prefetching algorithms such as sequential data ac- cess, strided access, and random access. Hermes also supports user de ned prefetching. In a way, the prefetcher becomes Hermes’ client for reading operations much like application cores are when writing data in DMSH. Messaging Service: is component is used to pass small mes- sages across the cluster of compute nodes. is 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 ush bu ered data of a certain le to the next layer or to PFS. I/O Clients: ese 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 bu ers, 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 le could be mapped into a temporary POSIX le in NVMe. e I/O clients give Hermes the exibility to ”talk” to several data destinations and manage the independent systems (e.g., memcpy for RAM, fwrite() for NVMe, MPI File write() for burst bu ers). System Pro ler: is component is a service outside the main library. It is designed to run once during the initialization. It performs a pro ling 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 con gured with. Using this information, the data placement engine can do a be er job when mapping data to di erent layers. Each system will have di erent hierarchy. Additionally, each hierarchy will demonstrate di erent performance characteristics. In our prototype implementation this component is external and results are manually injected to the con guration of the library. We plan to automate this process. Schema Parser: is component accepts a user-de ned bu ering schema and embeds it into the library. is schema is passed in a XML format and Hermes is con gured accordingly. For instance, if user chooses to aggressively bu er a certain dataset or le, then Hermes will prioritize this data higher up in the hierarchy and also the cache manager will get informed not to evict this speci c bu ered dataset. All this is possible because Hermes will use the user’s instructions to o er the best bu ering performance. In our prototype implementation schema parser is external and is planned to be automated in future versions of Hermes.

Applications Coordinator: is component is designed to o er support in a multiple-application environment. It manages the access to the shared layers of the hierarchy such as the burst bu ers. Its goal is to minimize interference between di erent applications sharing this layer. Additionally, it coordinates the ushing of the bu ers to achieve maximum I/O performance. More information on this component can be found in [29].

All the above components allow Hermes to o er a high perfor- mance I/O bu ering platform which is highly con gurable, easily pluggable to several applications, adaptable to certain system ar- chitectures, and feature-rich yet lightweight.

3.2 Hermes Bu ering Modes and Policies

3.2.1 Bu ering modes. Similar to other bu ering systems, Her- mes o ers several bu ering modes (i.e., con gurable by the user) to cover a wide range of di erent application needs such as I/O latency, fault tolerance, and data sharing: A. Persistent: in this mode, data bu ered in Hermes is also wri en to the PFS for permanent storage. We have designed two con gura- tions for this mode. 1) Synchronous: directs write I/O onto DMSH and also to the underlying permanent storage before con rming I/O completion to the client. is con guration 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, o ers strong data consistency, is ideal for data sharing between processes, and supports read-a er-write work- loads. However, it demonstrates the highest latency and lowest bandwidth for write operations since data directed to the bu ers also need to be wri en in the PFS. 2) Asynchronous: directs write I/O onto DMSH and completion is immediately con rmed to the client. e contents of bu ers are eventually wri en down to the permanent storage system. e trigger to ush bu ered data is con gurable and can be: i) per-operation, ushing is triggered at the end of current fwrite(), it also ushes all outstanding previous operations, ii) per- le, ushing is triggered upon calling fclose() of a given le (this is similar to Data Elevator approach), iii) on- exit, ushing is triggered upon application exit (this is similar to Datawarp approach), and iv) periodic, ushing is periodically trig- gered in the background (this is the default Hermes se ing). is con guration 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 a er writing to the bu ers. It also o ers eventual consistency since data are ushed 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 wri en down to the permanent storage. It is designed to o er a scratch space for fast temporary I/O. Upon application exit, Hermes deletes all bu ered data. is mode can be used for sce- narios such as quickly storing intermediate results, communication between processes, in-situ analysis and visualization. In case of bu ering node failures, application must restart. is mode o ers high bandwidth and low latency. Lastly, applications can reserve a speci c allocation (i.e., capacity on bu ers) for which data preser- vation is guaranteed by Hermes (similar to Datawarp reservations). ese allocations expire with the application lifetime. In case of bu er over ow, Hermes will transparently swap bu er contents

Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Bu ering System HPDC ’18, June 11–15, 2018, Tempe, AZ, USA

to the PFS much like memory pages are swapped to the disk by the OS. e mechanism was designed to o er some extra degree of exibility to Hermes. For example, let us assume that an application writes simulation results every 5 minutes. ese results are directly read from the bu ers by an analysis kernel which writes the nal result to the PFS for permanent storage. Simulation data can be deleted or overwri en a er 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 bu er space. To achieve this conditional overwriting of data, Hermes utilizes a

agging system to de ne the lifetime of bu ered data. C. Bypass: in this mode, as the name suggests, I/O is performed directly against the PFS e ectively bypassing Hermes. is mode resembles write-around cache designs.

3.2.2 Data placement policies. In DMSH systems, I/O can be bu ered to one or more layers of the hierarchy. ere are two main challenges: i) how and where in the hierarchy data are placed, ii) how and when do bu ers get ushed either in the next layer or all the way down to PFS. In Hermes, the rst challenge is addressed by the data placement engine (DPE) component and the second by the data organizer. We designed four di erent data placement policies to cover a wide variety of applications’ I/O access pa erns. Each policy is described by a dynamic programming optimization1 and follows the ow of Algorithm 1. e general idea of the algorithm is as follows. First, if the incoming data can t in the current layer’s remaining capacity, it places the data there (i.e., PlaceData()). In case it does not t, based on the constraint of each policy, it tries one of the following: a) solve again for next layer (i.e., skip()), b) place as much data as possible in the current layer and the rest in next (i.e., split()), and c) ush current layer and then place new incoming I/O (i.e., ush()). We implemented the DP algorithm using memoization techniques to minimize the overhead of the solution. We further provide a con guration 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. e DPE places data in the highest possible layer of DMSH in a top-down approach, starting from RAM, while balancing bandwidth, latency, and the capacity of each layer. e approach applies to all layers making the solution recursively op- timal in nature. e 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, e ectively maximizing the bandwidth. e data organizer moves data down periodically (or when triggered) to increase the available space in upper layers for future incoming I/O. Data move- ment between layers is performed asynchronously. is policy is the default Hermes con guration. B. Maximum Data Locality: this policy aims to maximize bu er utilization by simultaneously directing I/O to the entire DMSH. e DPE divides and places data to all layers of the hierarchy based on a data dispersion unit (e.g., chunks in HDF5, les in POSIX and independent MPI-IO, and portions of a le in collective MPI-IO). Furthermore, Hermes maintains a threshold based on the capacity ratio between the layers of the hierarchy. is ratio re ects on the

1Full mathematical formulation of each policy can be found in the Appendix.

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

1 Hermes-DPE(data request, DMSH layer); 2 if data can t in current layer then 3 PlaceData() ; // buffer data in this layer 4 else 5 MaxConstraint( // based on selected policy 6 - skip() ; // buffer in next layer 7 - split() ; // buffer in both current and next layers 8 - ush() ; // buffer in current layer after flushing 9 ); 10 end

relationship between each layer (e.g., system equipped with 32GB RAM, 512GB NVMe, and 2TB burst bu ers creates a capacity ratio of 1-16-64). e data placement in this policy accounts for both layer’s capacity and data’s spatial locality. e 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) e ectively maximizing the bu er utilization. Data movement between layers is performed asynchronously. is policy is ideal for work ows 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 le over another (e.g., metadata

les over data les). C. Hot-data: this policy aims to o er applications a fast cache for frequently accessed data (i.e., hot-data). e DPE places data in the hierarchy based on a hotness score that Hermes maintains for each le. is score encapsulates the access frequency of a le. Highest scored les will be placed higher up in DMSH since they are expected to be accessed more o en. is ensures that layers with lower latency and higher bandwidth will serve critical data such as metadata, index les, etc. e DPE also considers the overall le size to e ciently map data to each layer (i.e., smaller les bu ered in RAM whereas larger les in burst bu ers). e 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. e data organizer demotes or promotes data based on the hotness score and the data movement is performed asynchronously. is policy is ideal for work ows that demonstrate a spectrum of hot-cold data. D. User-de ned: this policy aims to support user-de ned bu er- ing schemas. Users are expected to submit an XML le with their preferred bu ering requirements. is le is parsed during ini- tialization by the schema parser component and used by the DPE to make data placement decisions. For instance, user can de ne certain les to always be in RAM (i.e., never get evicted), or which HDF5 chunks to get bu ered in NVMe etc.

3.3 Implementation Details

3.3.1 Node design. e new DMSH system architecture sug- gests that compute nodes may be equipped with one or more non- volatile storage device and share access to a burst bu er deployment. Hermes is designed to support all the new trends in system design. Figure 3 demonstrates Hermes node design. Each application core uses an I/O API (i.e., POSIX, MPI-IO, HDF5 etc.) which in turn

HPDC ’18, June 11–15, 2018, Tempe, AZ, USA Anthony Kougkas, Hariharan Devarajan, Xian-He Sun

Figure 3: Compute node design in Hermes.

is captured by Hermes. A dedicated core per node, called Node Manager, is exclusively used by Hermes services. Speci cally, this multi-threaded core is responsible for metadata management, data organization and movement between layers, messaging services between compute nodes (horizontal hierarchy), local memory man- agement such as placement of data in buckets, eviction policies, and nally prefetching. e ratio between application cores and the Hermes node manager is con gurable and is suggested to be around 64-to-1 (i.e., similar to I/O forwarding layer present in several su- percomputing 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.

3.3.2 Critical components. During I/O bu ering into DMSH, there are three critical operations: memory, metadata, and com- munication management. To achieve high-performance in each of these critical operations, Hermes incorporates several novel tech- nical innovations. As it can be seen in Figure 3, RAM is split into application memory and Hermes memory, which is further divided in bucket pool, MDM, and message queue. A. RAM management. We have designed a new memory man- agement system to o er fast and e cient use of main memory, a very crucial resource in any bu ering platform. Hermes stores data in buckets, an abstract notion of a data holder. Buckets have a con gurable xed size and consist of a collection of memory pages. All buckets are allocated during the bootstrapping of the system, creating a bucket pool. is allows Hermes to avoid the cost of per-request memory allocation (i.e., only pay the cost in the begin- ning before application starts), to be er control memory usage by avoiding expensive garbage collection, and to de ne the lifetime of memory allocations per application (i.e., re-use the same buckets a er data have been ushed down). Bucket pools are organized in four regions: available buckets, RAM cache, NVMe cache, burst bu ers cache. e bucket pool is managed by the bucket manager who is responsible to keep track of the status of each bucket (e.g., full - available). e bucket, as a unit of bu ering, is extremely critical to achieve high performance, low latency, and increases design exibility (e.g., be er eviction policies, hot data cache etc.).

Malloc TC-Malloc Hermes 3.5x106

d noces/snoitarepO2.5x101.5x10500000 3x102x101x106 6 6

6 6

0 64 128 512 1024 2048

Allocation size (KB)

Figure 4: RAM operations throughput.

We implemented Hermes’ memory management using MPI one- sided operations. Speci cally, buckets are placed in a shared dy- namic Remote Memory Access (RMA) window. is allows easier access to the buckets from any compute node and a be er global memory management. MPI-RMA implementations support RDMA- capable networks which further diminishes the CPU overhead. 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. e structure of a bucket includes an identi er (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 le stored in a non-volatile layer of the hierar- chy (con gured by user). We suggest placing this special le to the burst bu ers since if a compute node fails, the local NVMe device will become unavailable till the node is xed.

Figure 4 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. e test was conducted on our development machine that runs CentOS 7.1. In the test’s baseline, we intercept each fwrite(), allocate a memory bu er (i.e., malloc()), copy data from user’s bu er to the newly allocated space (i.e., memcpy()), and nally ush the bu er (i.e., free()) once the data are wri en 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 bu er to the acquired buckets. Once data are wri en to the disk, buckets are marked by the data organizer as available and no freeing is performed. As it can be seen in gure 4, 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. e pre-allocation and e cient management of the buckets and the lack of freeing of bu ers helped Hermes to maintain stable high performance. B. Metadata management. Any metadata service in distributed systems is subject to scalability and performance issues. Metadata in a bu ering platform like Hermes consist of data distribution information (e.g., which node, which layer in DMSH, which bucket,

Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Bu ering System HPDC ’18, June 11–15, 2018, Tempe, AZ, USA

50000

0 Custom(MPI) Redis Memcached Hermes

Figure 5: Metadata Manager throughput.

etc.) and maintenance of both user’s and internal le namespaces. Hermes’ metadata manager is distributed and aims to o er 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 speci c values. Our design uses two RMA windows: i) key window, which is indexed to support e cient querying and ii) value window, for data values. is practically allows any process to simply MPI Get() a speci c key and then fetch its respective value. We use a 2-way hashing: rst, the key is hashed to a speci c node and then into a value that resides on that node. e 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 e ectively 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 le 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 o er 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) le handler to le: maintains le handlers of opened les, {fh, lename}, ii) le to metadata prop- erties: maintains all typical le properties (e.g., permissions, own- ership, timestamps etc.,), { lename,{ lestat}}, iii) les to location in DMSH: maintains data distribution information, { lename,{(o set, size),(node,layer,type,identi er,freq)}}, and iv) node to current status: maintains information for each node’s current status such as remain- ing capacity, hot data access frequencies, etc., {node,(layer,size,...)}. ese maps allow fast queries and O(1) read/write MDM opera- tions without the need to execute separate services (e.g., a mem- cached server). Creation and update of metadata information is performed by using MPI EXCLUSIVE locks which ensures FIFO con- sistency. Read operations use a shared lock which o ers higher per- formance 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.,).

In Figure 5 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

350000

and answers queries from other processes. Upon receiving one, it d noces/snoitarepO300000 250000 200000

queues the operation, it spawns a thread to serve the operation, and it goes back to listening. e spawned thread removes the operation 150000 100000

from the queue and performs the operation. While this approach is feasible, it uses a dedicated core per node. Another approach is

Creations Updates Operation type

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. ere 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. ere is no dedicated core used. As it can be seen in Figure 5, 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 rst need to retrieve the metadata, update them, and then push them back. C. Messaging service. Many operations in Hermes involve com- munication between di erent compute nodes, bu ering nodes, and several other components. e 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 bu ers are ushed to the remote parallel le 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 a ributes, 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. ere is no need to employ listeners or other always-on services such as Apache ActiveMQ [49] or Ka a [30] leading to be er resource utilization. Additionally, we de ne 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 o er higher overall performance avoiding network bo lenecks and communication storms.

In Figure 6 we compare Hermes’ performance with a custom MPI-based solution, Memcached, and NATS. In this test, we is- sue 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

HPDC ’18, June 11–15, 2018, Tempe, AZ, USA Anthony Kougkas, Hariharan Devarajan, Xian-He Sun

0

Custom(MPI) NATS Memcached Hermes

Figure 6: Messaging Service throughput.

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 la er 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 gure 6, Her- mes outperforms the custom MPI-based messaging implementation by more than 12x. is 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. e 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 be er performance compared to the custom MPI-based one. Finally, NATS performance is really good with more than 300000 published messages per second. However, Hermes outper- forms NATS by more than 2x for publishing and more than 3x for subscribe operations. 3.4 Design Considerations In this subsection, we brie y discuss concerns regarding the design and features of any bu ering platform, especially one that supports a DMSH system such as Hermes. e 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 pro ler component which identi es and bench- marks 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 pa erns favorable to the each medium. Concern 2: How to avoid excessive network tra c? 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?

e several data placement policies of Hermes’ DPE provide tun- able performance guarantees for a variety of workloads. For low latency applications, Hermes can leverage the performance charac- teristics of each layer by placing data to the fastest possible layer. Additionally, our novel memory management ensures that data can be e ciently cached in RAM before ending up to their bu er. Concern 4: How to avoid possible bu er over ow?

Hermes’ Data Organizer component manages the capacities of the

900000 d noces/segasseM800000 700000 600000 500000

layers and moves data up and down the hierarchy (i.e., between the layers). In corner cases of over ow, Hermes provides explicit triggers to the data organizer to re-balance the layers and move 400000 300000

data based on the bu er capacity on each layer.

200000 100000

Publish Subscribe

Operation type

Concern 5: How to scale the bu er 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) de- ployments [57] to further extend the bu er capacity. B. Fault tolerance: Fault tolerance guarantees are based on the bu ering mode selected (i.e., sync, async). In case of asynchronous bu ering mode, bu ered data are wri en to a fault tolerant layer such as a PFS eventually which means for a small window of time bu er contents are sus- ceptible to failures. In our prototype implementation, bu ers are ushed 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 con gurable 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 bu ered 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. e model supported is single-writer, multiple-readers. Concern 2: Support of highly concurrent metadata operations? Upon opening a le, metadata are loaded from the PFS to the local RAM of the process that opened it. en, Hermes randomly selects two other nodes and replicates metadata there. We do this to in- crease the availability of the metadata info and avoid saturation of one node’s RAM. When another process wants to access the meta- data, 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. is 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 rst pro le his/her application using typical monitoring and pro ling tools, such as Darshan [9], 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 ex- periments. We rst explored how Hermes’ data placement policies handle di erent workloads and application characteristics using synthetic benchmarks. We then compare Hermes with state-of-the- art bu ering platforms, namely Data Elevator and Cray’s DataWarp, using real applications. As performance metric, we use the overall execution time in seconds which we further divide to: i) time to write/read to/from bu ers, and ii) time to ush bu ers to PFS. Com- putation time is excluded since it is the same among all systems.

Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Bu ering System HPDC ’18, June 11–15, 2018, Tempe, AZ, USA

**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 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

2000) ces(e mitl larevO1800 1600 1400 1200 1000 800 600 400 0 Write Flush 20 μs 55-66 μs

200 BaselinDeataData-Intensive EDleav

taWHearrmp

esHBeW

rmesDL

BaselinDeataWorkload EDleav Balanced taWHearrmp

esHBetype W rmesDL

BaselinDCompute-Intensive eataEDleav taWHearrmp

esHBeW

rmesDL

Figure 7: Testbed speci cations.

Figure 8: Benchmark: Alternating Compute-I/O phases.

MPI Collective I/O) that various analysis modules read back. We As reference, we include a baseline of no bu ering in which data

used 16 time steps for both simulations resulting to total I/O of 1TB. are wri en/read directly to/from the PFS. We run all tests ten times and we report the average time.

4.2 Experimental Results Hardware: All experiments were conducted on Chameleon [13].

4.2.1 Synthetic Benchmarks. Our synthetic benchmark is highly More speci cally, we used the bare metal con guration with 32

tunable to generate workloads that can stress the bu ering system client nodes (i.e., up to 1024 MPI ranks), 8 burst bu er nodes, and

under various use-cases. We designed two test-cases to evaluate 16 PFS storage nodes. Each node has a dual Intel(R) Xeon(R) CPU

Hermes’ data placement policies. E5-2670 v3 running at 2.30GHz with a total of 48 cores, and 128 GB

Alternating Compute-I/O phases: In this test, each process rst RAM. Each burst bu er node is equipped with an SSD drive and

performs some computations (emulated by sleep() calls) and then each PFS node with an HDD. We emulated one NVMe device per

writes 64MB in a le-per-process fashion. We repeat this pa ern client node by deploying a DRAM-based le system (i.e., RAMDISK)

16 times with 1024 processes resulting in 1TB total I/O size. We and imposing latency and bandwidth penalties to match the actual

vary the ratio of computation over I/O time to emulate three dis- NVMe performance [20, 52, 55]. In order to correctly calculate

tinct types of applications: data-intensive, compute-intensive, and the added latency and lowered bandwidth, we captured the perfor-

balanced. We assume that all data wri en to the bu ers need to mance characteristics of real NVMe devices present in the hierarchy

be also wri en to the disk-based remote PFS. erefore, Hermes appliances of Chameleon. Figure 7 lists all the hardware speci -

is con gured in persistent asynchronous mode. We measure the cations and performance measurements. Lastly, to be er capture

overall time spent in I/O, in seconds, which consists of write-time the architecture of a modern supercomputer, we setup our cluster

and ush-time. Figure 8 shows the results. As it can be seen, the topology as follows: all 32 client nodes and 8 burst bu ers are in-

baseline writes directly to PFS (i.e., no ush-time) and maintains terconnected with 56Gbps In niband network and the 16 storage

stable write performance regardless of the computation-I/O ratio. nodes are connected to the rest via a 10Gbps Ethernet network.

In Data Elevator and DataWarp, data are wri en to the burst bu ers So ware: e operating system of the cluster is CentOS 7.1, the

resulting to similar write-time between them. e di erence in per- MPI version is Mpich 3.2, the PFS we used is OrangeFS 2.9.6, the

formance comes from data ushing. Data Elevator overlaps ushing in-memory key-value stores are Memcached 1.4.36 and Redis 4.0.6,

with computation phases, and thus, as the computation-I/O ratio and lastly the distributed queue we used is NATS Server 1.0.4.

increases, ush-time decreases (i.e., ushing is hidden behind com- Applications: We evaluate Hermes using our own synthetic bench-

putation). On the other hand, DataWarp ushes data only once the mark that emulates common scienti c application workloads such

application nishes and demonstrates stable ush-time regardless as alternation between computation - I/O phases, read -a er-write,

of the computation-I/O ratio. In Hermes, data are wri en in all lay- read-once, read-many etc. It uses POSIX-IO to issue requests to

ers of the DMSH (i.e., RAM, NVMe, and burst bu ers in our system). the le system and operates in a typical le-per-process pa ern.

We evaluate both MaxBW and MaxLocality data placement policies We also use two real science applications: Vector Particle-In-Cell

since they bu er data di erently. MaxBW places data in a top-down (VPIC), a general purpose simulation code for modeling kinetic

fashion. It starts with RAM for the rst iterations of the test, and plasmas in spatial multi-dimensions, and Hardware Accelerated

once this layer is full, it rst moves data down to NVMe to create Cosmology Code (HACC), a cosmological simulation that studies

space in RAM and then places the incoming iteration in RAM. On the formation of structure in collisionless uids under the in uence

the other hand, MaxLocality uses layers concurrently. It writes the of gravity in an expanding universe. Both of these simulations per-

rst iterations in RAM and once this layer is full it goes on to the form computations and produce output les periodically that need

next without any data movement between layers. It is clear that for to be persisted in PFS. Also, both demonstrate a periodic behavior

data-intensive applications where the rate of incoming I/O is high, with time steps (i.e., iterations) that include the checkpoint and

MaxBW’s data movement between layers imposes some perfor- restart as well as the analysis outputs produced by the simulations.

mance losses, and thus, MaxLocality’s write performance is slightly At the end of each step, VPIC writes a single HDF5 le containing

higher. As the computation-I/O ratio increases however, MaxBW properties of 8 million particles. VPIC tends to be extremely I/O

can overlap data movement between layers with computations. intensive (i.e., write-only, write-heavy), since the portion of com-

erefore, for compute-intensive workloads, MaxBW outperforms putation is small. In contrast, HACC has read-a er-write workload

MaxLocality by 4x in write-time since it ensures that incoming I/O where, at every step, simulation writes out a single shared le (i.e.,

can be wri en in RAM. For ushing, both policies leverage any

HPDC ’18, June 11–15, 2018, Tempe, AZ, USA Anthony Kougkas, Hariharan Devarajan, Xian-He Sun

1600Write Read 1400 1200 1000 1800 1600 1400 1200 1000 800 800 600 600 400 400 200 200 0 BaselinDeataEDleRead-Once

av taWHearrmp

esHBeW

rmesHD

BaselinDeataRead-Many EDleav

taWHearrmp

esHBeW

rmx4 esHD

BaselinDeataRead-Many EDleav

taWHearrmp

eHsBeW

rmx16 esHD

0 BaselinDeataEleDvataWHaerp

rmesBW

BaselinDeataEleDvataWHaerp

rmesBW

BaselinDeataEleDvataWHaerp

rmesBW

Figure 9: Benchmark: Repetitive Read operations.

computation time available to asynchronously ush bu er contents to PFS, similarly with Data Elevator. However, Hermes ushes all layers of the DMSH concurrently which decreases ush-time signi cantly. In summary, in this test Hermes o ers 8x and 2x higher write performance when compared to No Bu ering baseline and state-of-the-art bu ering platforms respectively. Repetitive Read operations: In this test, the benchmark is con- gured to create a write-once, read-many workload. Each process rst writes 32MB in a le-per-process approach and then reads back 32MB of data (not necessarily the same data). We have 16 phases of this pa ern 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. is pat- tern resembles workloads where portions of data such as metadata information, indices of les, etc., are frequently accessed creating a data hotness spectrum. In this test, we assume that bu ers are used as scratch space (i.e., temporary I/O), and thus, Hermes is con gured in non-persistent mode. e total time, in seconds, is divided into write-time and read-time. As it can be seen in Figure 9, 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 wri en/read to/from the burst bu ers respectively. is results to a considerable performance improvement over the baseline. Since repetitive read operations are treated as new, it shows stable performance across di erent work- loads. In contrast, Hermes implements a HotData data placement policy to o er 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 signi cant performance boost for Read-many x4 and x16. On the other hand, MaxBW, while o ering 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 o ers 38x and 11x higher read performance when compared to No Bu ering baseline and state-of-the-art bu ering platforms respectively.

4.2.2 Real Applications. To test our system under real applica- tions workload, we con gured 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: is application demonstrates a write-only I/O access pat- tern where at the end of each time step, each process writes data to an HDF5 le. During this evaluation we executed the application

Write Flush ) ces(e mitl larevO) ces(e mitl larevOWorkload type

256

Number of 512 processes 1024 Figure 10: I/O Bu ering performance with VPIC-IO.

1400

) ces(e mitl larevO1200 1000 800 600 400 200 Write Read Flush

0 BaselinDeataEleDvat256 aWHaerp

rmesBW

BaseNumber linDeataEleDvaof t512 aWHprocesses

aerp rmesBW

BaselinDeataEleDva1024 taWHaerp

rmesBW

Figure 11: I/O Bu ering performance with HACC-IO.

for 16 time steps. We strong scaled the application from 256 to 1024 total ranks and we measured the total time. In Figure 10 we report only the I/O time which consists of write-time (i.e., what the application experiences) and ush-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. is is approximately a 2.5x improvement over the baseline. How- ever, due to the higher bandwidth of the DMSH, Hermes’ write performance is 5x and 2x higher than the baseline and the two bu ering platforms we tested, respectively. When considering data ushing, Data Elevator overlaps small computations between each time step and ushes the contents of burst bu ers in 1115 seconds whereas DataWarp ushes everything at the end in 1274 seconds. In contrast, Hermes leverages the computations but also the concur- rency of the DMSH to ush all bu ered data to PFS in 637 seconds. In summary, in this test, Hermes outperformed the baseline and state-of-the-art bu ering platforms by 40% and 85% respectively. HACC: is application demonstrates a read-a er-write I/O access pa ern where during each time step, each process reads back data previously wri en 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 11 we report only the I/O time which con- sists of write-time, read-time, and ush-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. is is approximately a 3.7x improvement over the baseline. However, when considering data ushing, Data Elevator completed the test in 773 and DataWarp in 985 seconds e ectively reducing the total improvement to 1.6x and 1.3x respectively. In contrast, Hermes completed the entire test in 494 seconds showcas- ing the potential of a DMSH system. e performance improvement

Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Bu ering System HPDC ’18, June 11–15, 2018, Tempe, AZ, USA

is substantial when compared to No Bu ering baseline with 7.5x faster I/O operations. Hermes outperformed Data Elevator and DataWarp by 2x due to higher bandwidth of the DMSH. 5 RELATED WORK New hardware technologies have been developed and can be used to build new memory and storage hierarchies using non-volatile mem- ory (NVRAM) such as phase-change memory (PCM) [42], mem- ristors [50], and Flash memory [12]. Flash-based SSD technology has been widely studied [24], characterized [21], and evaluated for di erent application types [3, 14]. Researchers also advocate the use of shared bu er technologies, such as burst bu ers [6], to accel- erate I/O. Existing work has considered NVMe devices as a viable solution for I/O staging [25, 26]. Caul eld proposed Moneta [11], an architecture with NVRAM as an I/O device for HPC applica- tions. Ekel extended Moneta with a real PCM device to understand the performance implications of using NVRAM [2]. Dong studied NVRAM for HPC application checkpointing [18]. Kannan studied NVRAM for I/O intensive benchmarks in Cloud environments [26]. Wang proposed BurstMem [53], a technology for optimizing I/O using burst bu ers. Sato et al., show how the burst bu ers can boost performance of checkpointing tasks by 20x [46].

Active Bu ers [35, 36] exploits one-sided communication for I/O processors to fetch data from compute processors’ bu ers and performs actual writing in the background while computation con- tinues. IOLite [40], proposes a single shared memory per-node for leveraging inter-process communication and bu ering of I/O. Such an approach led to 40% boost in performance. Nitzberg [39] proposes collective bu ering algorithms for improving I/O per- formance by 100x on IBM SP2 at NASA Ames Research Center. PLFS [4] remaps an application s preferred data layout into one which is optimized for the underlying le system.

While all the above work emphasizes the bene ts of using each technology individually, none introduced a complete I/O bu ering platform that leverages the DMSH. e closest work to Hermes is Data Elevator [17], a new system that transparently moves data in a hierarchical system. e authors focused on systems equipped with burst bu ers and demonstrated a 4x improvement over other state-of-the-art burst bu er management systems such as Cray’s Datawarp [16]. 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 bu ers. Furthermore, Hermes extends bu ering into remote resources and tackles data movement to a more complicated landscape of I/O-capable devices.

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 signi cantly 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 signi cant burden currently le 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 bu ering system. Hermes enables, manages, and

supervises I/O bu ering into the DMSH and o ers a bu ering plat- form 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 uti- lization, abstracting data movement, maximizing performance, and supporting a wide range of scienti c applications and domains. We have presented three novel data placement policies to e ciently utilize all layers of the new memory and storage hierarchy as well as three novel techniques to perform memory, metadata, and com- munication management in hierarchical bu ering systems. Our evaluation results prove Hermes’ sound design and show a 8x im- provement compared to systems without I/O bu ering support. Additionally, Hermes outperforms by more than 2x state-of-the-art bu ering platforms such as Data Elevator and Cray’s Datawarp.

ACKNOWLEDGMENT

is material is based upon work supported by the National Science Foundation under Grants no. CCF-1744317, CNS-1526887, and CNS- 0751200.

REFERENCES

[1] Sean Ahern, Sadaf R Alam, Mark R Fahey, Rebecca J Hartman-Baker, Richard F Barre , Ricky A Kendall, Douglas B Kothe, Richard T Mills, Ramanan Sankaran, Arnold N ar- rington, et al. 2007. Scienti c application requirements for leadership computing at the ex- ascale. Technical Report. Oak Ridge National Laboratory (ORNL); Center for Computational Sciences. [2] Ameen Akel, Adrian M Caul eld, Todor I Mollov, Rajesh K Gupta, and Steven Swanson. 2011.

Onyx: A Prototype Phase Change Memory Storage Array. HotStorage 1 (2011), 1. [3] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 1–14. [4] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: a checkpoint lesystem for parallel applications. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, 21. [5] John Bent, Gary Grider, Bre Ke ering, Adam Manzanares, Meghan McClelland, Aaron Tor- res, and Alfred Torrez. 2012. Storage challenges at Los Alamos National Lab. In Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on. IEEE, 1–5. [6] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Ma Bryson, Joaquin Correa, Glenn K Lockwood, Vakho Tsulaia, et al. 2016. Ac- celerating science with the NERSC burst bu er early user program. CUG2016 Proceedings (2016). [7] D Brown, Paul Messina, D Keyes, J Morrison, R Lucas, J Shalf, P Beckman, R Brightwell, A Geist, J Ve er, et al. 2010. Scienti c grand challenges: Crosscu ing technologies for comput- ing at the exascale. O ce of Science, US Department of Energy, February (2010), 2–4. [8] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and improving computational science storage access through continuous characterization. ACM Transactions on Storage (TOS) 7, 3 (2011), 8. [9] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 characterization of petascale I/O workloads. In Cluster Computing and Workshops, 2009. CLUSTER’09. IEEE International Conference on. IEEE, 1–10. [10] Adrian M Caul eld, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K Gupta, Allan Snavely, and Steven Swanson. 2010. Understanding the impact of emerging non-volatile memories on high-performance, I/O-intensive computing. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 1–11. [11] Adrian M Caul eld, Arup De, Joel Coburn, Todor I Mollow, Rajesh K Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In Proceedings of the 2010 43rd Annual IEEE/ACM International Sym- posium on Microarchitecture. IEEE Computer Society, 385–395. [12] Adrian M Caul eld, Laura M Grupp, and Steven Swanson. 2009. Gordon: using ash memory to build fast, power-e cient clusters for data-intensive applications. ACM Sigplan Notices 44, 3 (2009), 217–228. [13] Chameleon.org. 2017. Chameleon system. (2017). h ps://www.chameleoncloud.org/about/

chameleon/ [14] Shimin Chen. 2009. FlashLogging: exploiting ash devices for synchronous logging perfor- mance. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. ACM, 73–86. [15] S Conway and C Dekate. [n. d.]. High-Performance Data Analysis: HPC Meets Big Data. ([n.

d.]). h p://www.hpcuserforum.com/presentations/tuscon2013/IDCHPDABigDataHPC.pdf [16] CRAY Inc. 2017. Datawarp technology. (2017). h p://www.cray.com/sites/default/ les/

resources/CrayXC40-DataWarp.pdf [17] Bin Dong, Suren Byna, Kesheng Wu, Hans Johansen, Je rey N Johnson, Noel Keen, et al. 2016. Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. In High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on. IEEE, 152– 161.

HPDC ’18, June 11–15, 2018, Tempe, AZ, USA Anthony Kougkas, Hariharan Devarajan, Xian-He Sun

[18] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. 2009. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exas- cale systems. In High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on. IEEE, 1–12. [19] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. 2011. e international exascale so ware project roadmap. e international journal of high perfor- mance computing applications 25, 1 (2011), 3–60. [20] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Ra- jesh Sankaran, and Je Jackson. 2014. System so ware for persistent memory. In Proceedings of the Ninth European Conference on Computer Systems. ACM, 15. [21] Kaoutar El Maghraoui, Gokul Kandiraju, Joefon Jann, and Pratap Pa naik. 2010. Modeling and simulating ash based solid-state disks for operating systems. In Proceedings of the rst joint WOSP/SIPEW international conference on Performance engineering. ACM, 15–26. [22] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A le format and I/O library for high performance computing applications. In Proceedings of Supercomputing, Vol. 99. 5–33. [23] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. 2009. e fourth paradigm: data-intensive

scienti c discovery. Vol. 1. Microso research Redmond, WA. [24] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G Shin. 2011. FAST: ick Application

Launch on Solid-State Drives.. In FAST. 259–272. [25] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. 2009. Perfor- mance trade-o s in using nvram write bu er for ash memory-based storage devices. IEEE Trans. Comput. 58, 6 (2009), 744–758. [26] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, Dejan Milojicic, and Vanish Talwar. 2011. Using active NVRAM for I/O staging. In Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities. ACM, 15–22. [27] Youngjae Kim, Raghul Gunasekaran, Galen M Shipman, David A Dillow, Zhe Zhang, and Bradley W Se lemyer. 2010. Workload characterization of a leadership class storage cluster. In Petascale Data Storage Workshop (PDSW), 2010 5th. IEEE, 1–5. [28] Rob Kitchin. 2014. Big Data, new epistemologies and paradigm shi s. Big Data & Society 1,

1 (2014), 2053951714528481. [29] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2017. Harmonia: An Interference- Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Bu ers. Technical Report. Illinois Insitute of Technology. [30] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Ka a: A distributed messaging system for

log processing. In Proceedings of the NetDB. 1–7. [31] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev akur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A high-performance scienti c I/O interface. In Supercomputing, 2003 ACM/IEEE Conference. IEEE, 39–39. [32] Glenn K Lockwood, Damian Hazen, incey Koziol, RS Canon, Katie Antypas, Jan Balewski, Nicholas Balthaser, Wahid Bhimji, James Bo s, Je Broughton, et al. 2017. Storage 2020: A Vision for the Future of HPC Storage. Technical Report. NERSC. [33] Jay F Lofstead, Sco Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flex- ible io and integration for scienti c codes through the adaptable io system (ADIOS). In Pro- ceedings of the 6th international workshop on Challenges of large applications in distributed environments. ACM, 15–24. [34] Los Alamos National Lab. [n. d.]. Trinity specs. ([n. d.]). h p://www.lanl.gov/projects/

trinity/speci cations.php [35] Xiaosong Ma, Marianne Winsle , Jonghyun Lee, and Shengke Yu. 2001. Faster collective out- put through active bu ering. In Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM. IEEE, 8–pp. [36] Xiaosong Ma, Marianne Winsle , Jonghyun Lee, and Shengke Yu. 2003. Improving MPI-IO output performance with active bu ering plus threads. In Parallel and Distributed Processing Symposium, 2003. Proceedings. International. IEEE, 10–pp. [37] Ningfang Mi, Alma Riska, Qi Zhang, Evgenia Smirni, and Erik Riedel. 2009. E cient man-

agement of idleness in storage systems. ACM Transactions on Storage (TOS) 5, 2 (2009), 4. [38] NERSC. [n. d.]. Cori system burst bu er design. ([n. d.]). h ps://www.nersc.gov/users/

computational-systems/cori/burst-bu er/ [39] Bill Nitzberg and Virginia Lo. 1997. Collective bu ering: Improving parallel I/O performance. In High Performance Distributed Computing, 1997. Proceedings. e Sixth IEEE International Symposium on. IEEE, 148–157. [40] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. 2000. IO-Lite: a uni ed I/O bu ering

and caching system. ACM Transactions on Computer Systems (TOCS) 18, 1 (2000), 37–66. [41] JB Peter. 2004. e Lustre storage architecture. Cluster File Systems, Inc (2004). [42] Moinuddin K reshi, 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. [43] Daniel A Reed and Jack Dongarra. 2015. Exascale computing and big data. Commun. ACM

58, 7 (2015), 56–68. [44] David Reinsel, John Gantz, and John Rydning. 2017. Data Age 2025: e Evolution of Data

to Life-Critical. Don t Focus on Big Data (2017). [45] Robert B Ross, Rajeev akur, et al. 2000. PVFS: A parallel le system for Linux clusters. In

Proceedings of the 4th annual Linux showcase and conference. 391–430. [46] Kento Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R De Supinski, Naoya Maruyama, and Satoshi Matsuoka. 2014. A user-level in niband-based le system and check- point strategy for burst bu ers. In Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on. IEEE, 21–30. [47] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large

Computing Clusters.. In FAST, Vol. 2. [48] John Shalf, Sudip Dosanjh, and John Morrison. 2010. Exascale computing technology chal- lenges. In International Conference on High Performance Computing for Computational Science. Springer, 1–25. [49] Bruce Snyder, Dejan Bosanac, and Rob Davies. 2017. Introduction to apache ActiveMQ. Active

MQ in Action (2017), 6–16. [50] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. e

missing memristor found. nature 453, 7191 (2008), 80–83.

[51] Rajeev akur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In Frontiers of Massively Parallel Computation, 1999. Frontiers’ 99. e Seventh Sym- posium on the. IEEE, 182–189. [52] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. artz: A light- weight performance emulator for persistent memory so ware. In Proceedings of the 16th An- nual Middleware Conference. ACM, 37–49. [53] Teng Wang, Sarp Oral, Yandong Wang, Brad Se lemyer, Sco Atchley, and Weikuan Yu. 2014. Burstmem: A high-performance burst bu er system for scienti c applications. In Big Data (Big Data), 2014 IEEE International Conference on. IEEE, 71–79. [54] Whi , Justin L. 2017. Oak Ridge Leadership Computing Facility: Summit and Beyond. (2017). h ps://indico.cern.ch/event/618513/contributions/2527318/a achments/1437236/ 2210560/SummitProjectOverview{ }jlw.pdf [55] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime Data Management on Non-

Volatile Memory-based Heterogeneous Main Memory. arXiv preprint arXiv:1705.00249 (2017). [56] Bing Xie, Yezhou Huang, Je rey S Chase, Jong Youl Choi, Sco Klasky, Jay Lofstead, and Sarp Oral. 2017. Predicting output performance of a petascale supercomputer. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. ACM, 181–192. [57] Dawid Zawislak, Brian Toonen, William Allcock, Silvio Rizzi, Joseph Insley, Venkatram Vish- wanath, and Michael E Papka. 2016. Early investigations into using a remote ram pool with the Vl3 visualization framework. InIn Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), Workshop on. IEEE, 23–28.

APPENDIX A. Maximum Application Bandwidth (MaxBW):

DPEMaxBW (s, Ci) =

 (s/BWi ) ∗ Ai min

, s ≤ Ci

(1) 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 Mo e(min size,dest) triggers data organizer to recursively move at least min size data to dest layer. B. Maximum Data Locality: DPEMaxLocalit (s, d, Li, Ri) =



DPE(s, Mo DPE(Ce(s i, − CCi) i, + i DPE(s + 1) Ci+1)

+ DPE(s, − Ci, CCi+1)

i )) , s > Ci

(s/BWi ) ∗ d , Li & s ≤ Ri min

(2) 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 ReOr anize(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:

DPEHotData (s, h, Hi, Ci) =



( DPE(s, (s/BWd, i) L∗ i+1, (d + R1)

i+1))

, !Li & s ≤ Ri

min

DPE(RReOr i, d, anize(s LDPE(s, i, Ri) − + Rd, DPE(s i) L+ i+1, DPE(s, − Ri+1)

Ri, d, d, LLi+1, i, Ri)) Ri+1)

, s > Ri

DPE(s, min

DPE(CE iict , h, (s H− i, CCi, i (s/C) h, + i h DPE(s + i − )/BW 1) 1, + HDPE(s, − i+1C, iC, h i+1− h, )

1, HHi, i+1Ci , ))

Ci+1)

, h ≥ Hi & s ≤ Ci min

, h ≥ Hi & s > Ci (DPE(s, h DPE(s, h, (3) where s is the request size, h is the le’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 E ict(min size,score,dest) is a function that triggers data organizer to recursively move at leastmin size data to thedest layer with score hotness.

+ H1, i+1H, i, Ci+1Ci ) ))

, h < Hi & s ≤ Ci

min

(DPE(Ci, h + 1, DPE(s, Hi, Ci) h, + DPE(s Hi+1, C− i+1C)

i, h, Hi+1, Ci+1)

), h < Hi & s > Ci