UNITY: Unified Memory and File Space*

Terry Jones, Michael J. Brim,
Geoffroy Vallee,
Benjamin Mayer, Aaron Welch,
Tonglin Li
Oak Ridge National Laboratory
P.O. Box 2008, MS 6164
Oak Ridge, Tennessee 37831-6164
trj@ornl.gov

Michael Lang,
Latchesar Ionkov,
Douglas Otstott
Los Alamos National Laboratory
P.O. Box 1663
Los Alamos, New Mexico 87544-0600
mlang@lanl.gov

Ada Gavrilovska,
Greg Eisenhauer,
Thaleia Doudali,
Pradeep Fernando
Georgia Institute of Technology
266 Ferst Drive
Atlanta, Georgia 30332-0765
ada@cc.gatech.edu

ABSTRACT

This paper describes the vision for *UNITY*, a new high-performance computing focused data storage abstraction that places the entire memory hierarchy, including both traditionally separated memory- and file-based data storage, into one storage continuum. Through the use of a novel API and a set of services centered around a smart runtime system, *UNITY* is able to provide a number of valuable and interesting benefits. The unified storage space provides a scalable and resilient data environment that dynamically manages the mapping of data onto available resources based on multiple factors, including desired persistence and energy budget considerations. By eliminating the need for high-performance computing domain scientists to develop architecture-dependent optimizations for rapidly evolving data storage technologies, *UNITY* addresses both ease-of-use and performance.

CCS CONCEPTS

• **Software and its engineering** → **Runtime environments**;
Ultra-large-scale systems; • **Hardware** → *Emerging interfaces*;

KEYWORDS

memory hierarchy, runtime systems, automated data management

ACM Reference format:

Terry Jones, Michael J. Brim, Geoffroy Vallee, Benjamin Mayer, Aaron Welch, Tonglin Li, Michael Lang, Latchesar Ionkov, Douglas Otstott, and Ada Gavrilovska, Greg Eisenhauer, Thaleia Doudali, Pradeep Fernando. 2017.

*This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>). This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, through the Storage Systems and Input/Output (SSIO) for Extreme Scale Science Program under Award Number ERKJ312.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

ROSS '17, Washington, DC, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5086-0/17/06...\$15.00
DOI: 10.1145/3095770.3095776

UNITY: Unified Memory and File Space. In *Proceedings of ROSS '17, Washington, DC, USA, June 27, 2017*, 8 pages.
DOI: 10.1145/3095770.3095776

1 INTRODUCTION

High-performance computing (HPC) practitioners have known of memory-induced performance issues at least as far back as William Wulf's 1995 paper on the "memory wall" [36]. Today, these problems remain despite two decades of computer science advances, and the situation is exacerbated by new challenges including energy budgets and deeper, more complex memory hierarchies. What was once a few distinct tiers is now a continuum of memory and storage technologies with complicated trade-offs in performance, capacity, and resilience. The consequence of more complexity to application developers is unsatisfactory. With each new technology or move to a new HPC platform, application changes are necessary to fully exploit the available capabilities. We need to revolutionize the way we address data challenges. Fortunately, we have a past success story to draw on: memory usage was revolutionized in the late 1960s when application developers shifted from explicitly managing overlays to automated virtual memory. Once hardware support was added, the abstraction benefits of a virtual memory interface were combined with performance benefits over even the best hand-tuned overlay scheme. This paper advocates a new shift toward increased automation and advanced system software capabilities to manage complex data storage hierarchies—a system that incorporates more recent concerns such as energy budgets and resiliency trade-offs.

The *UNITY* [1] project envisions a unified data environment where applications, tools, and system services easily and efficiently access and share data regardless of its placement within the memory and storage hierarchy. The primary design goals of *UNITY* are (1) a unified application abstraction for data that integrates emerging memory and storage technologies, (2) a simplified application interface for data objects, and (3) the ability for applications to influence the runtime to meet requirements such as persistence, energy budget, latency, and scalability. In this work we present our vision for *UNITY*, including its architectural design to achieve our goals. Our overall objective is to design, implement, and evaluate a new distributed runtime that unifies the traditionally distinct application views of memory- and file-based data storage into a single scalable and resilient data environment. This simplifies an application's interaction with its native data objects and hides the complexity of new emerging storage architectures. Moreover, the abstractions provided to enable automation also address traditional

concerns such as ease-of-use and performance while providing the flexibility to adapt to new constraints including energy budgets and resilience.

We believe memory architectures have grown too complex to manage with today's interfaces. These problematic interfaces are simply too limited in their ability to capture sufficient information to resolve tradeoffs associated with data placement. *UNITY*'s design allows one to identify what one wishes to optimize: are we optimizing for bandwidth, or latency, or energy efficiency, or resilience, or some other factor?

The data abstraction provided by *UNITY* increases performance portability by eliminating requirements on HPC domain scientists to develop architecture-dependent optimizations for rapidly evolving data storage. A smart runtime system manages memory and storage resources and attempts to optimize data placement and layout. Instead of explicitly moving data between memory and storage, users describe the precise subsets of data they need for their computation. The runtime can locate the necessary data, retrieve it (if the location is remote), and convert it to the appropriate in-memory format. Since users are not required to define and use on-disk data formats, the runtime has the freedom to choose layouts to improve the performance of all applications that access the data. *UNITY* is designed under the premise that the tedious task of managing voluminous data objects dynamically is best performed by computers, rather than domain scientists. The runtime is responsible for performing dynamic data placement based on application-provided annotations while weighing complex trade-offs related to latency, capacity, access patterns, resilience, and energy budgets. The *UNITY* data abstraction also permits the runtime to alter data storage layouts as the access patterns change from data production to data analysis.

This work makes two key contributions:

- **An architectural description of *UNITY*, a collection of services that provide data placement and movement within complex memory hierarchies based on user-provided annotations**
- **A simplified data abstraction for applications that unifies memory and storage, decouples application data layout from physical media layout, and provides explicit data versioning for consistent data sharing without global synchronization**

The paper is organized as follows. A Background section describes the prevailing circumstances that triggered our work. Next, an approach section lays out the high-level design goals, while the *UNITY* design and architecture section describes the key aspects of our unified data abstraction and the associated runtime system architecture. An implementation section describes our current prototype software system that provides our unified data environment for HPC platforms. This is followed by a related works section, and our summary.

2 BACKGROUND

2.1 Memory hierarchy advances

New technologies and architectural designs are resulting in a large amount of new and emerging complex memory technologies on

HPC systems. Examples of these are: Multi-channel DRAM (MC-DRAM on Intel Xeon Phi), PCI-e nonvolatile memory (NVM), SATA NVM, phase-change memory, memristor, and 3Dxpoint. Additionally, other technologies that are “near-node” accessible, such as memory attached to a PCI-e switch and network attached memories, are expected to show up in exascale memory designs.

Current practice for incorporating the features of these new technologies is dependent on vendor-provided libraries or ad hoc solutions rather than a uniform abstraction that provides the needed features and portability. The ad hoc approach is already insufficient and future memory technologies will only compound the problem by adding additional proprietary APIs.

2.2 Evolving sharing considerations

Data movement, both on node and between nodes, is one of the most energy-expensive and performance-limiting components of using HPC supercomputers. To decrease data movement, applications are employing new techniques including tightly coupled physics codes that are co-scheduled on the same nodes and in situ data analysis and visualization that reduce the amount of data saved on persistent storage. The usage model for HPC machines is evolving from sequences of simulation and analysis tasks that communicate via long-term storage, toward a more dynamic approach where applications consist of complex combinations of coupled codes, and in situ data services plus tools. New practices associated with realtime experimentation and observational data are driving more complex data sharing between supercomputer simulations and experimental data produced by national science facilities such as accelerators, colliders, and light sources [7]. Projected limitations in the I/O system motivate integrating simulation and analysis, coupling complex physics codes, and developing fully integrated application workflows [9, 29, 37].

These new approaches increase the necessity of improved data sharing APIs. Historically, processes running on the same node used shared memory techniques, complemented by double- and multi-buffer algorithms to improve performance. New techniques, such as TCASM [27] further improve the local sharing by eliminating most of the need of synchronization between applications. Across nodes, applications use three main techniques for sharing data: files on parallel filesystem, explicit message passing, and partitioned global address space (PGAS) approaches like SHMEM and GASNet. High performance through distributed file-based sharing is particularly difficult because of the shared file pointer limitations in POSIX; this is partially addressed by strategies such as MPI-IO [16], but we desire a more general solution. Moreover, file-based sharing makes it easy to share data between loosely coupled applications, but it does not support sharing data in memory and forces additional copy of the data on persistent storage. The other two approaches require much tighter coupling of the applications and do not support NVM (although there are extensions looking into adding support for that [19]).

2.3 Managing for data resilience and durability

The presence of NVM, with order of magnitude faster access times compared to current persistent storage devices, and order of magnitude larger capacity than traditional DRAM, offers unique opportunities to accelerate services for resilience and reliability, relying on the ability to persist application state. However, naively using existing solutions with future nonvolatile RAMs (NVRAMs) will not yield desirable results. The details of these technologies will vary across vendors, but when compared to DRAM, they are broadly characterized with expensive write latencies, high write energy, and limited bandwidth and endurance. In the past, our team has developed several optimizations to traditional techniques for checkpointing and persistence to permit them to better leverage future NVRAMs. These include avoiding data serialization and use of memory-based APIs, use of shadow buffers, predictive methods for pre-copying, and energy-aware persistence [21, 22]. We leverage these methods in the design and implementation of *UNITY*.

3 APPROACH

3.1 Unified data environment

UNITY allows data to be shared between applications and processes within an application regardless of where it is stored in the unified memory and storage hierarchy. It provides support for processes accessing data that is mapped into other process' address space.

3.2 Separating policy from strategy details

UNITY aims to hide the complexities of the underlying memory substrate from applications, while presenting to applications interfaces using which they can specify their desired data properties (e.g., concerning performance, consistency, or persistence) or their broader management objectives (e.g., maximizing performance, resilience, or energy efficiency).

3.3 Intelligently managing tradeoffs

The design of *UNITY* encompasses a range of mechanisms and abstractions to capture the cost and benefit of its underlying data management operations, thereby automating the evaluation of complex decision tradeoffs about its operation, and to offer support for diverse data management policies. Some of these are briefly described in the description of the *UNITY* runtime.

3.4 Co-designed from inception

Co-design refers to a computer system design process where scientific problem requirements influence architecture design and technology and constraints inform formulation and design of algorithms and software [2]. Co-design is of paramount importance for HPC practitioners since both the complexity of the computing platforms and scientific simulations is drastically increasing over time, virtually creating a gap between application improvements (in terms of scalability and performance) and hardware potential performance. By having a dialog between domain scientists, computer scientists, and hardware designers/vendors, initiatives such as the US Department of Energy's (DOE's) co-design program witness a gain in terms of scalability and performance, as well as an increase

of scientific output from research programs. Furthermore, this co-design approach usually allows the domain scientists to switch focus from technical details to benefit from modern hardware and runtimes to the science that is critical to their simulations.

In keeping with the co-design approach, Accelerated Climate Modeling for Energy (ACME) has been selected as a "platform" to highlight the benefits of *UNITY*. ACME is the DOE climate flagship project, with allocations on all of the major US DOE open science supercomputers.

4 UNITY DESIGN AND ARCHITECTURE

The main components of *UNITY* can be divided into two types: on-node services and distributed services. The on-node services manage the data locally available on the node, and make decisions about data placement in local resources (Figure 1). The global services include a nameserver that keeps track of the abstract objects that are defined, as well as the physical location of the data (Figure 2). Another service allows global optimization of fragment's placement and layout. *UNITY* is designed to be programming model agnostic; for example it is well-suited to both message passing and asynchronous task models.

Figure 1: *UNITY* node architecture.

Objects represent abstract datasets that contain semantically consistent data irrespective of how the data is produced or how the simulation is partitioned across the supercomputer components. Data for an object can be produced by multiple processes of a single application, or even by many applications.

Fragments define a subset of an object. They are the smallest data entity that *UNITY* applications can read or modify. In addition to specifying the parts of an object that are of interest to a process, fragments also define how the data is represented in the process' memory. For example, the fragment defines the element ordering in multidimensional arrays, as well as the data alignment. In that

Figure 2: UNITY system architecture.

sense, the fragment is a concrete data layout for a subset of the abstract object.

4.1 Nomenclature and Design Components

To ensure data consistency and enable data sharing, *UNITY* defines versions for both objects and fragments. Versions are 64-bit values that increase during the object's life cycle. Normally, the processes that compose a distributed application run a loop and in each iteration they use the data produced by the previous iteration to compute the new data. When they share or store data, they use some global synchronization mechanism in order to ensure that the data is consistent and is produced by the same iteration. *UNITY* makes the sharing easier by explicitly defining a version number for each data fragment. When a process produces new data for a fragment, it publishes the data as a new version. An object's version is not explicitly modified. It is defined as the highest version of the fragments that cover the whole object. Other processes or services can wait for a specific version to become available and progress when it does. This approach saves applications from costly global synchronization operations, and it removes demands of applications to know, in advance, which process produced the data required. For example, in an application that partitions some simulated space, a process can publish a new version for the subspace it is simulating, then wait for the halo regions it needs to be published by other ranks. This application would not need to employ a global barrier, or know the the ranks of the processes that produced the data it needed.

UNITY does not keep a single version of its objects and fragments. Rather, it expects that multiple versions will be available as the application makes progress. When an object is created, the user can specify flags that guide *UNITY* as to how many versions should

be kept for an object, as well as if any objects need to be persisted once an application completes its execution.

The *UNITY* *runtime*s execute on each node and manage the local resources. They keep track of the fragments that are located in RAM and locally attached NVM. The *runtime* has two distinct interfaces. The local interface is used by the applications running on the node and allows them to communicate with the *nameserver*. This interface can be used to create objects and fragments, attach to fragments, and publish fragments that the application already has attached. The *runtime* uses TCASM [4] to share memory with the application in a consistent manner. In the current implementation, when a process requests a fragment to be attached, the *runtime* creates a file in the shared memory filesystem and populates it with data retrieved by fragments that overlap with the requested fragment. When the *runtime* returns, it passes the name of the shared memory file back to the process so it can *mmap* the file. When the application publishes a new version of the data by using TCASM, the *runtime* ensures that it has a copy of it by calling *mmap*. The remote interface of the *runtime* is used by the *nameserver* and other *runtime*s to access data from the local fragments.

The *nameserver* coordinates the work of the *runtime*s and keeps the global data of *UNITY*. For each object, it keeps descriptions of the data types and abstract entities it contains, as well as descriptions of the subsets that define each fragment. The *nameserver* is also aware of the location of each fragment's versions. Using this information it can generate a list of locations and transformation rules that are needed to populate a newly created fragment. The *runtime*s use this functionality to implement support to attach a new fragment to the application. Additional performance opportunities can be obtained through use of NVM and hybrid-memory-aware allocators and object stores [20].

4.2 Programming abstraction and API

In this section, we describe how an application might use *UNITY*. In the following example, we assume a 2D mesh constructed as a 20 x 20 array of integers. For the object creation, the *nameserver*

ensures that the object name (in our case "A") is unique, and checks if its description is valid. The object is created with the following call:

```
objectDescription = "var a[20,20] int32"
CreateObject("A", objectDescription, flags);
```

The flags parameter to the *CreateObject* allow for attributes to be associated with the object. *UNITY* uses these to stipulate how latency, capacity, resiliency, and energy budget tradeoffs should be resolved. For instance, if the application indicates that this object should be persisted, *UNITY* will attempt to consolidate and write all fragments of the object to stable storage at certain predetermined intervals. Obviously, the optimal configuration (which storage media should be leveraged for persistence, how often *UNITY* should attempt to checkpoint the object, etc.) will vary depending on the application in question and the system that *UNITY* is deployed on. It is our intention that *UNITY* will soon have the ability to query the system for the presence of relevant hardware and achieve near-optimal storage performance without explicit instruction.

By defining a rank's local data as a subset of the global object, *UNITY* can infer the key attributes of the dataset, specifically the type and the size of the region to be produced. What used to be a typical variable declaration is now replaced with these *UNITY* calls:

```
String workFragment =
    "var w[i:5, j:5] = a[xstart,jystart]";
void *data = AttachFragment("A", workFragment, 0);
```

As each command is received by the *runtime*, it will pass essential metadata to the *nameserver*.

The *nameserver* and the *runtime* index the fragments within objects. As mentioned in the previous section, each *runtime* is responsible for indexing its locally created fragments, as well as registering them with the *nameserver*. In this way, the *nameserver* is aware of each fragment's location on the network by referencing the *runtimes* that are indexing them.

On receiving the `AttachFragment` command, the *runtime* will request a new fragment ID from the *nameserver* and provide the fragment description and the network location of the *runtime*. The *nameserver* will use the descriptions of both the fragment and object to determine the size of the fragment in question. This information is returned to the *runtime* with the fragment ID that the *nameserver* used to index the fragment. This ID is stored by the application library for use in other *UNITY* operations.

As a result of the `AttachFragment` operation, the application receives a pointer to the area in the memory that contains the data for a specific version of the requested fragment. The first version, version 0, is initialized with default values for its data.

UNITY includes a function that an application may use to notify the system that a new version has been reached: `PublishFragment`.

```
for(step = 0; step < end; step++) {
    // computations using the data area
    PublishFragment(data, step + 1);
}
```

When the *runtime* receives a `PublishFragment` request it creates an internal record for the new version and uses TCASM to attach it to the data. It then notifies the *nameserver* that a new version of the fragment is available at its location, so if there is another *runtime* waiting for the data it can proceed. The *runtime* keeps the data in the format that it is produced in.

The concept of versions is important for many use cases, such as checkpoint/restart and interprocess communication. For instance, if the value written to a cell is dependent on values stored in its neighbors, then cells that fall on the boundaries between ranks may require data which is in the domain of a different process and may reside on a remote machine. Retrieving this data requires interprocess communication, potentially via the network.

The design for *UNITY* provides mechanisms to save unnecessary communication for common simulation components such as "ghost cells." By including a specific version as a parameter for `AttachFragment`, as well the explicit readonly status indicated in the description string, the application tells its *runtime* these calls should not return until the source data required to construct these fragments is made available by another rank.

5 IMPLEMENTATION

5.1 Existing software

UNITY leverages several open-source projects. Many of the components, the *nameserver* for example, can easily be replaced by other key/value stores. The current set of components was chosen for functionality, and developers familiarity, to enable rapid development.

Hop [17] is a framework for building distributed key/value stores. It allows specification of different consistency and replication levels, and by allowing the assembly of different implementations of the interface, it provides the right balance of consistency, scalability, and availability to create custom system services. Hop supports versioning of entries that fits nicely with the *UNITY* data model. We used Hop for implementing both the *nameserver* and the *runtime*.

DRepl [18] decouples the data layouts used by data producers and consumers from the way data is saved on the storage system. DRepl provides a powerful language for describing objects and their fragments, as well as functionality for generating and executing optimized transformation rules that convert the data between fragments. We use DRepl as the basis for marshaling data between the application and physical layouts in *UNITY*.

TCASM [27] is a shared-memory copy-on-write kernel that allows asynchronous progress between the publisher of data and one or more observer(s). TCASM consists of kernel-level modifications to multiple memory management functions, the most significant being the `msync` system call. We introduced a new flag, `MS_UPDATE`, that allows applications to indicate a desire to sync changes made to privately mapped data back to the `mmap'd` file. Normally this is not permitted: privately mapped files are write-protected and thus result in a copy-on-write, leaving any changes visible only to the writing process. However, TCASM's changes to `msync` include functionality to find these copied-on-write memory pages and insert them into the actual `mmap'd` file. TCASM takes this one step further by saving any pages that are mapped by other processes, so that the changes are not immediately visible. In this way, TCASM exposes a sort of publish-and-subscribe mechanism with built-in autoversioning. *UNITY* uses this TCASM to support its versioning and to allow asynchronous progress of applications.

SNOflake is a scalable network overlay infrastructure that supports distributed applications and services spanning multiple network domains. SNOflake employs a ring-of-trees overlay architecture wherein a tree overlay is deployed on each network domain, and an interdomain ring connects the roots of the domain trees. This architecture permits scalable overlays that cover entire HPC computing centers, including multiple compute, service, and storage networks. Distributed services built on SNOflake are provided with advanced communication capabilities including subscription-based data streaming and point-to-point messaging between overlay endpoints. Similar to MRNet [31], data streams support both multicast and reduction via in-network data filtering and aggregation.

5.2 Design components

The *UNITY* runtime plays a key role in driving the placement and migration of data elements across the distributed memory hierarchy. Orchestrating the data placement and movement operations must be carried out in a manner that reconciles a range of tradeoffs with

respect to application performance and overall efficiency of the underlying resources, and the runtime intelligence lies precisely in the ability to capture, reason about, and act on relevant information to maximize the desired priority metrics.

First, concerning the use of the aggregate capacity of the memory substrate, our goal is to (1) maximize the use of application-near fast memory capacity for application critical data (e.g., DRAM), while at the same time (2) reducing the overheads associated with dynamic data monitoring and dynamic data access characterization only when these can be outweighed by commensurate benefits. Current state-of-the-art approaches rely on detailed a priori application profiles to establish the relative priority of all of the application data structures [12, 28, 33]. However, such approaches have limited applicability for complex applications, applications exhibiting data-dependent variability in their execution, or requiring cross-application-component data exchanges. Instead, our goal is to identify data components most critical for application performance (e.g., in terms of their number or type of accesses or sizes), and to characterize how they contribute to the application's sensitivity toward different memory characteristics. In this manner, the runtime can delegate noncritical data to slower/farther memories (e.g., NVRAM), and focus on expending runtime resources to further maximize the fast memory capacity usage (e.g., by performing operations related to hotness tracking, dynamic data migration, etc.) for critical data. The initial approach for realizing this goal is based on establishing set of guiding rules that can help the developer, or, in the future, the toolchain, annotate application critical data and high-sensitivity program regions. Noncritical data is more aggressively directed toward the larger (though slower) capacity tier.

This tier is also a natural choice for buffering shared data exchanged across application components because of its large capacity and lower possibility for any backpressure on the data generating simulation components. One critical tradeoff, however, is that given the relative higher costs of NVRAM writes, naively writing out data for placement in this tier can incur write costs, which can be reduced or even eliminated with data compression techniques. It is the runtime's responsibility to understand and act on the tradeoffs among write access and data movement reduction, versus any additional computation cycles that must be expanded to carry out the (de-)compression operations or to deal with the additional complexities of ensuring consistent program state.

In addition, particularly regarding the use of the NVM resources for application data with persistence requirements, the runtime is presented with another set of tradeoffs. To provide guarantees for persistent data for services such as checkpoint/restart, the runtime must aggressively move data into the persistent storage, with minimum cost in the critical path of the application generating the persistent snapshot (checkpoint state). With parallel application processes generating this state fairly in sync, this presents a challenge for the NVRAM bandwidth capacity, which is already expected to be limited for a number of technology-related reasons (e.g., write energy). We reconcile this by taking advantage of the combined aggregate, local and remote memory capacity and accompanying bandwidth. The resulting technique, described in detail in [13], splits the output, partially writes portion of it out to NVRAM, and temporarily writes out a portion to DRAM staging

buffers. The fault-tolerance of the DRAM-resident data is achieved through other techniques, specifically replication. Ultimately all data is migrated to the target NVRAM location, but such migrations can be performed asynchronously, outside of the critical path. Despite the increase data movement requirements, the approach still yields gains in execution time, and improves the overall energy efficiency of the system. The runtime remains responsible for dynamically assisting the overall resource availability and application demands in order to parametrize the execution of the technique, including the split ratio, or the details of realizing the persistence of the staging buffers, and it does this given target optimization metrics, such as gains in execution time, energy efficiency, or reliability guarantees. In addition, we integrate and further extend additional optimizations to reduce the overall data movement requirements, those taking place in the critical path of persistence operations, or both [13, 21], and will further consider methods for integrating energy awareness leveraging our earlier work [22].

By leveraging Hop for the *nameserver*, *UNITY* realizes an optimized key/value store with enough generality to specify our required consistency and replication levels. Currently the *nameserver* is a single process that our scalability experiments show will scale well up to tens of thousands of processes. Hop is designed to be used as a distributed key/value store and we are investigating distributed implementations of our *nameserver* for resiliency and increased scalability in the final product.

When *UNITY*'s global data placement requires movement of data between host storage locations, the SNOflake overlay is used to dynamically establish a data transfer between the source and destination hosts. If the hosts reside on the same network, a direct connection between the hosts is created to perform the transfer. Where available, the common communications interface [5] layer underlying SNOflake will leverage high-bandwidth remote memory access operations supported by the network. For transfers between hosts from separate network domains, the overlay is used to generate a multihop transfer route. Usually, an interdomain transfer requires three hops, one within the source network (i.e., source host to tree root), one within the interdomain ring network (i.e., tree root to tree root), and one within the destination network (i.e., tree root to destination host).

6 RELATED WORK

Existing efforts to improve storage and memory subsystems for HPC follow three main approaches. The first approach is to expose more implementation details and parameters to higher layers (i.e., applications and libraries) and require developers to choose the appropriate values for their application. The second approach is to mimic existing storage interfaces while providing transparent methods, such as caching, replication, and tiering, that may improve performance. Finally, the third approach is to introduce high-level data abstractions, typically via an API or runtime, that completely hide storage details and rely on embedded intelligence to derive an optimized storage strategy that meets user requirements.

Storage Exposure

The first approach, exposure of storage details, is typical of parallel file systems such as Lustre [32] and PVFS [30], which expose

low-level data striping decisions to users. In *UNITY*, we favor exposing high-level semantic decisions, such as desired persistence and availability, rather than low-level storage details.

Another example of exposure is the original burst buffer concept [23] and related approaches [3, 26, 37] that employ nodes positioned (logically) between compute nodes and persistent storage, often populated with high-performance nonvolatile storage devices. When an application running on compute nodes enters a state of high-write bandwidth (e.g., during a checkpoint), the data is sent to the burst buffer nodes instead of the regular storage servers because their high-performance devices can service requests much faster, allowing the application to return to execution sooner. Then, the burst buffers are free to slowly drain the massive data back to persistent storage. This is similar to the *UNITY* paradigm of publishing and persisting, where a process can notify *UNITY* that a section of data is ready to be persisted and quickly continue without an I/O operation, and *UNITY* is free to persist this data locally or remotely without blocking the application, thanks to TCASM's shared memory semantics. Unlike burst buffers that are designed specifically for large streaming writes, *UNITY* is a general purpose data management runtime that can efficiently support read-oriented workloads such as analytics, while still being able to incorporate burst buffers as part of the unified storage it manages.

Storage Mimicry

PMFS [11] provides a highly optimized filesystem to NVM and implements fast memory-mapped I/O by avoiding data being copied from NVRAM to DRAM. Tiered storage refers to systems that provide some form of intelligent migration to decide how individual pieces of data should be stored in an attempt to optimize the cost/performance relationship [15]. Typically, the storage system would be exposed to the application as a single device (or file system). Applications may read and write from this device as usual, and the system will track aggregate usage statistics to determine media storage. Data moved between devices for tiered storage will be marked for removal within the destination. Campaign storage is a form of tiered storage that is focused on long-term (i.e., archival) storage of massive datasets. MarFS [8] is a campaign storage system that presents a single filesystem namespace that spans backend storage systems, including cloud storage, object storage and tape archives, with different performance characteristics and capacities. Like *UNITY*, these systems are completely transparent to applications. They hide details of storage devices, which are managed by intelligent algorithms. However, unlike *UNITY*, applications still use (virtual) files and devices, and thus must block to write. There is also no simplified interface for data sharing through tiered or campaign storage.

High-level Data Abstractions

Multiple projects provide high-level abstractions to address the challenges of emerging on-node persistent memory technologies. NV-Heaps [10] provides an object system for NVRAM and a set of common data types such as search trees, hash tables, and sparse graphs. Mnemosyne [35] provides compiler support and primitives for automatic persistence of application data to NVRAM. *UNITY* adopts a similar high-level approach that uses an intelligent runtime to manage fragments of objects within persistent memory.

HDF5 [14] is a library that provides a rich data model for describing scientific data. It allows definitions of datasets, groups, datatypes, and metadata objects, as well as relations between them. HDF5 is highly customizable and provides many opportunities for custom plug-ins that solve specific performance problems, although the multilayered design makes it hard to implement new features. Another disadvantage of HDF5 is that, as a library linked to the application, it cannot exploit opportunities for data optimization once the application completes.

The Adaptable I/O System (ADIOS) [25] provides a simplified storage abstraction intended to eliminate the need for explicit I/O calls in scientific applications. Users provide an external XML file that defines data elements, their types, and the intended method of processing. ADIOS-based applications use the provided APIs to manage data (persisting, sharing, etc.) instead of calls specific to each medium. In this way, once an application has been ported to ADIOS, there is no need to modify the application when switching to a different transport method. ADIOS and *UNITY* share numerous similarities. Hiding storage details behind a high-level API is a key design goal for both. ADIOS and *UNITY* also share facilitating application coupling with in situ analytics and visualization. Both approaches allow users to trigger application-specific checkpointing and to employ a self-describing data format (i.e., BP for ADIOS, and DRepl for *UNITY*). The biggest difference is that ADIOS is a runtime that is compiled into the application; there is no ADIOS process that runs persistently on any node. *UNITY*, however, is a distributed system that is always running. By leveraging the DRepl dataset descriptions, *UNITY* can create new dataset views during application execution, whereas ADIOS must use the data layouts specified in the static XML file. Finally, ADIOS still requires the user to indicate specific storage methods in the XML configuration. In contrast, *UNITY* is free to dynamically adapt its storage behavior based on up-to-date knowledge of resource utilization.

Legion [6, 34] is a data-centric programming model and runtime for executing applications on distributed parallel systems. Legion maps each computation step to its data dependencies (i.e., the input of one step is the output of another). Legion then associates each region of data produced by the application with a specific computational step. Legion's "software out-of-order" scheduler is analogous to a hardware out-of-order scheduler, where a data region is treated like a hardware register and instructions have dependencies based on when and if they read or write from the same register(s). Data may be replicated across multiple regions. The goal is to optimize process and data placement to minimize the cost of communication and data retrieval over the course of execution. *UNITY* strives to be programming model independent, and it provides data abstractions and a runtime that is compatible with many existing programming models.

DAOS [24] is a project that attempts to solve many storage issues in the HPC environment. DAOS extends the standard storage APIs by providing a union of most of the solutions from the past. It defines a richer set of data objects: blobs that are similar to POSIX files, multidimensional arrays that are useful for HPC application developers, containers that combine other types, and key/value stores that are specialized containers. DAOS also defines multiple techniques to ensure consistency: distributed transactions, epochs, and

data versioning. In *UNITY*, data objects are not as specialized as in DAOS, but they are versioned through publish-subscribe semantics.

7 SUMMARY

We have described our vision for *UNITY*, a new HPC-focused data storage abstraction that places the entire memory hierarchy, including both traditionally separated memory- and file-based data storage, into one storage continuum. The design is motivated by increasingly complex memory hierarchies and the associated effort required by domain scientists to realize the available features and characteristics of these new systems. In particular, *UNITY* provides a unified and simplified application abstraction for data that integrates emerging memory and storage technologies, a simplified application interface for data objects, and the ability to honor requested a set of predefined data attributes including persistence, energy budgets, latency, and scalability. The abstraction provided by *UNITY* increases performance portability via a smart runtime system. By eliminating requirements on HPC domain scientist to develop architecture-dependent optimizations for rapidly evolving data storage, *UNITY* addresses both ease-of-use and performance.

REFERENCES

- [1] 2016. The *UNITY* Project - Merged Memory and Storage I/O. <http://www.unity-ssio.org>. (2016). Accessed: 2017-03-05.
- [2] 2017. ASCR Co-Design Centers. <https://science.energy.gov/ascr/research/scidac/co-design/>. (2017). Accessed: 2017-03-05.
- [3] Hasan Abbasi, Jay Lofstead, Fang Zheng, Karsten Schwan, Matthew Wolf, and Scott Klasky. 2009. Extending i/o through high performance data services. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 1–10.
- [4] Hakan Akkan, Latchesar Ionkov, and Michael Lang. 2013. Transparently consistent asynchronous shared memory. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 6.
- [5] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich. 2011. The Common Communication Interface (CCI). In *2011 IEEE 19th Annual Symposium on High Performance Interconnects*. 51–60. DOI: <https://doi.org/10.1109/HOTI.2011.17>
- [6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. <http://dl.acm.org/citation.cfm?id=2388996.2389086>
- [7] E. Wes Bethel and Martin Greenwald (eds.). 2016. *Report of the DOE Workshop on Management, Analysis, and Visualization of Experimental and Observational Data: The Convergence of Data and Computing*. Technical Report LBNL-1005155. Lawrence Berkeley National Laboratory, Berkeley, CA, USA, 94720. Also available as https://science.energy.gov/~media/ascr/pdf/programdocuments/docs/ascr-eod-workshop-2015-report_160524.pdf.
- [8] David John Bonnie and Kyle E. Lamb. 2016. *MarFS: A Scalable Near-POSIX Name Space over Cloud Objects and New Features*. DOI: <https://doi.org/10.2172/1333128>
- [9] Ron Brightwell, Ron Oldfield, Arthur B Maccabe, and David E Bernholdt. 2013. Hobbes: Composition and virtualization as the foundations of an extreme-scale OS/R. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2.
- [10] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *SIGPLAN Not.* 46, 3 (March 2011), 105–118. DOI: <https://doi.org/10.1145/1961296.1950380>
- [11] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. DOI: <https://doi.org/10.1145/2592798.2592814>
- [12] Subramanya R. Dulloor, Amitabha Roy, and others. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 15, 16 pages. DOI: <https://doi.org/10.1145/2901318.2901344>
- [13] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan. 2016. Phoenix: Memory Speed HPC I/O with NVMM. In *Proceedings of the IEEE conference on High Performance Computing, Data, and Analytics (HiPC)*. Hyderabad, India.
- [14] Mike Folk, Gerd Heber, Quincey Kozio, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 36–47.
- [15] Larry Freeman. 2012. What's Old Is New Again - Storage Tiering. http://www.snia.org/sites/default/education/tutorials/2012/spring/storman/LarryFreeman_What_Old_Is_New_Again.pdf. (2012).
- [16] William Gropp, Ewing Lusk, and Rajeev Thakur. 1999. *Using MPI-2: Advanced features of the message-passing interface*. MIT press.
- [17] Latchesar Ionkov and Michael Lang. 2015. Hop: Elastic Consistency for Exascale Data Stores. In *Inter. Conf. on High Performance Computing*. Springer, 291–306.
- [18] Latchesar Ionkov, Michael Lang, and Carlos Maltzahn. 2013. Drepl: optimizing access to application data for analysis and visualization. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 1–11.
- [19] Latchesar Ionkov and Ginger Young. 2014. Asymmetric Memory Extension for Openshmem. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM, New York, NY, USA, Article 22, 2 pages. DOI: <https://doi.org/10.1145/2676870.2676887>
- [20] Sudarsun K., A. Gavrilovska, and K. Schwan. 2016. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 13, 16 pages. DOI: <https://doi.org/10.1145/2901318.2901325>
- [21] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. 2013. Optimizing Checkpoints Using NVMM as Virtual Memory. In *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2013)*.
- [22] Sudarsun Kannan, Moinuddin Qureshi, Ada Gavrilovska, and Karsten Schwan. 2016. Energy Aware Persistence: Reducing Energy Overheads of Memory-based Persistence in NVMMs. In *25th International Conference on Parallel Architectures and Compilation Techniques (PACT'16)*.
- [23] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 1–11.
- [24] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Kozio, John Bent, and Eric Barton. 2016. DAOS and friends: a proposal for an exascale storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 50.
- [25] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. 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*. ACM, 15–24.
- [26] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. 2008. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 9.
- [27] D. Otstott, N. Evans, L. Ionkov, M. Zhao, and M. Lang. 2014. Enabling composite applications through an asynchronous shared memory interface. In *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 219–224.
- [28] Antonio J. PeAa and Pavan Balaji. 2014. Toward the efficient use of multiple explicitly managed memory subsystems. In *Proc. IEEE International International Conference on Cluster Computing (CLUSTER'14)*. 123–131.
- [29] David Rogers, Kenneth D Moreland, Ron A Oldfield, and Nathan D Fabian. 2013. *Data co-processing for extreme scale analysis level II ASC milestone (4745)*. Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).
- [30] R. Ross, R. Thakur, and others. 2000. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conf*. 391–430.
- [31] P. C. Roth, D. C. Arnold, and B. P. Miller. 2003. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Supercomputing, 2003 ACM/IEEE Conference*. 21–21. DOI: <https://doi.org/10.1145/1048935.1050172>
- [32] Philip Schwan. 2003. Lustre: Building a file system for 1000-node clusters. In *2003 Linux Symposium*.
- [33] Du Shen, Xu Liu, and Felix Xiaozhu Lin. 2016. Characterizing Emerging Heterogeneous Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 13–23. DOI: <https://doi.org/10.1145/2926697.2926702>
- [34] Sean Treichler, Michael Bauer, and Alex Aiken. 2013. Language Support for Dynamic, Hierarchical Data Partitioning. *SIGPLAN Not.* 48, 10 (Oct. 2013), 495–514. DOI: <https://doi.org/10.1145/2544173.2509545>
- [35] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104. DOI: <https://doi.org/10.1145/2248487.1950379>
- [36] Wm A Wolf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.
- [37] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. 2010. PreDataA-preparatory data analytics on peta-scale machines. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.