Automatic and Transparent I/O Optimization With Storage Integrated Application Runtime Support

Noah Watkins
UC Santa Cruz
jayhawk@soe.ucsc.edu

Zhihao Jia
Stanford University
zhihao@cs.stanford.edu

Galen Shipman
LANL
gshipman@lanl.gov

Carlos Maltzahn
UC Santa Cruz
carlosm@soe.ucsc.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Pat McCormick
LANL
pat@lanl.gov

ABSTRACT

Traditionally storage has not been part of a programming model's semantics and is added only as an I/O library interface. As a result, programming models, languages, and storage systems are limited in the optimizations they can perform for I/O operations, as the semantics of the I/O library is typically at the level of transfers of blocks of uninterpreted bits, with no accompanying knowledge of how those bits are used by the application. For many HPC applications where I/O operations for analyzing and checkpointing large data sets are a non-negligible portion of the overall execution time, such a "know nothing" I/O design has negative performance implications.

We propose an alternative design where the I/O semantics are integrated as part of the programming model, and a common data model is used throughout the entire memory and storage hierarchy enabling storage and application level co-optimizations. We demonstrate these ideas through the integration of storage services within the Legion [2] runtime and present preliminary results demonstrating the integration.

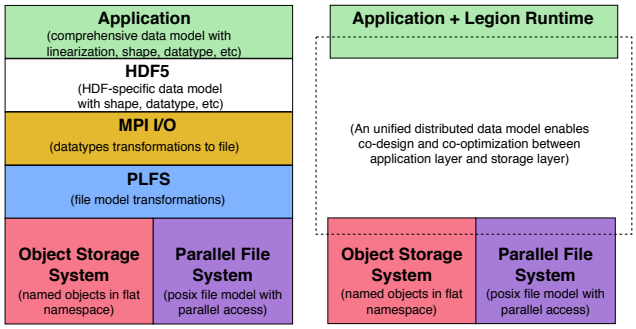


Figure 1: (a) In a contemporary I/O stack each layer uses a distinct data model. (b) Our proposed architecture uses a unified data model and run-time to enable system-wide co-design and co-optimization strategies

.

1. INTRODUCTION

Persistent I/O performance is a critical factor in HPC application efficiency, but applications today rarely achieve a meaningful percentage of the peak bandwidth of a storage system. While over-provisioning of storage resources has allowed inefficiencies to be masked, next-generation exascale systems will likely make such techniques economically infeasible.

The performance of storage systems and applications are tightly coupled, influenced by hardware and software configuration, as well as the behavior of constantly changing workloads. Despite this co-dependence, applications and storage systems have traditionally been developed in isolation, reducing opportunities for co-optimization to an afterthought, often through the use of non-portable hints that are either not easily configured or insufficiently expressive. Auto-tuning techniques have been used to tune system configurations, but these tools are often limited by interfaces that restrict optimization techniques [5, 4]. What is lacking in current system designs are interfaces that permit applications and storage systems to co-optimize behavior by exposing system-wide requirements and intent. In this paper, we propose the creation and integration of semantically rich I/O interfaces with next-generation distributed run-time systems that together utilize a common data model throughout the entire software stack, enabling co-optimization strategies that can be applied transparently to applications. Our work extends one previous effort that has looked at integrating persistent storage within an HPC programming system [7]; our approach allows for runtime (as opposed to compile-time) decision making and much more general control over data movement and layout.

The predominant approach to improving I/O performance for HPC applications has been through the introduction of I/O middleware libraries. Illustrated in Figure 1a, libraries such as HDF5 provide portable application-level data models, while others transparently implement I/O methods that use a variety of optimization strategies [11, 10, 8, 1]. However, as I/O flows through the opaque interfaces of the soft-

PDSW 2015 November 15-20, 2015 Austin, TX, USA

ware stack, application semantics are lost. For instance, PLFS performs transparent I/O transformations to optimize for checkpoint workloads, but in the process it completely re-writes the I/O stream rendering the dataset unrecognizable without consulting an index, potentially complicating the process of in-situ and offline analyses that have been tuned for alternative application data layouts. Further exacerbating the problem are the economic realities that require storage systems to often be deployed as organizationally shared resources with virtually no information regarding application access patterns or intent, preventing inter-workload optimizations that may be effective at mitigating the performance impact of resource contention. Thus, offline tuning and hard-coded optimization strategies may be difficult to achieve or ineffective because of cross-traffic from other workloads, and failures cannot be predicted. One alternative approach that has been proposed is to introduce a low-capacity, high-performance buffer-cache layer into the storage hierarchy [6, 9] that can absorb I/O bursts and reorganize I/O for optimal utilization of bulk storage. However, in general this type of extreme over-provisioning is not affordable as systems grow in scale. The solution we propose is a matter of co-design in which over-provisioning is replaced by intelligent, automatic management of the memory and storage hierarchy.

We demonstrate these ideas using the Legion run-time system [3], a task-based runtime with a fully distributed memory model that decouples the logical data model from the physical layout allowing the runtime to manage data dependencies and data placement transparent to the application all while respecting application consistency requirements. Shown in Figure 1b, we propose a new structure in which the Legion runtime replaces existing middleware layers by reproducing salient I/O optimizations and maintaining a consistent data model across the entire storage and memory hierarchy..

2. BACKGROUND

We now provide background information and motivate our work, briefly discussing common I/O optimization techniques, and the Legion programming model and runtime system that forms the basis for our prototype implementation.

2.1 Motivating Example

Consider the introduction of additional storage tiers to a typical HPC environment, where each tier may differ in capacity, bandwidth, and latency. Such an expanded storage hierarchy presents many opportunities for sophisticated storage strategies such as hiding latency, intelligently handling low-memory situations, performing I/O staging, and offloading data-intensive compute tasks. Unfortunately it can be difficult for applications to fully exploit the storage hierarchy when data must be managed explicitly by application developers.

For example, an application running on a system with a deep storage hierarchy may integrate knowledge about analysis tasks into its checkpointing strategy by storing one component of application state (e.g., a field associated with a grid point) on a fast tier composed of SSDs for a pending visualization workflow, while the remaining grid fields are placed in a capacity tier for resilience. The data management challenge involved in this example is difficult to solve. In current systems, the application would be required to split the data structure into multiple files, store the data in separate namespaces, and manage consistency and future tier migration. While middleware is capable of performing tasks such as a complex data mapping and sharding, the data management tasks required to track asynchronous updates to application state across a heterogeneous memory and storage hierarchy while conforming to the consistency requirements of the application are beyond the scope of current I/O libraries that provide I/O optimizations or flexible data serialization strategies. If such complex data management is embedded in the structure of an application, migrating to a new system may require invasive changes to the application that prevent adoption in the first place.

This example provides context for the type of data management challenges that will exist in next-generation systems, but we also consider a broad class of common strategies for improving I/O performance that we describe next.

2.2 Data Management and I/O Optimization

We now provide a brief overview of some common strategies for improving persistent I/O performance. These strategies are beneficial when they can be achieved, but performance portability can be a challenge, making adoption of new techniques less attractive.

I/O Parallelism. I/O parallelism is often achieved by *aligning* data, which guarantees that concurrent tasks are accessing disjoint parts of the data. But it can be challenging for an application to achieve good parallel I/O performance because the file abstraction hides physical properties that may allow applications to align I/O accesses and avoid inadvertently invoking expensive locking and consistency protocols. Applications may use *hints* provided by the storage system to align data within a file but hints are non-portable, and aligning unstructured data may be difficult. Sharding (e.g. N-N checkpoint, in which case each of N clients writes to its own file) is one way to avoid these alignment issues, but sharding is rarely used because data management is difficult.

Latency Hiding. Performing I/O asynchronously is a common technique to hide latency by overlapping I/O and computation. Applications written in low-level languages hard-code these optimizations, implicitly embedding performance assumptions that may be invalid on different architectures, or even from one run to the next. Finding and exploiting opportunities for latency hiding techniques can be difficult and quickly add complexity to application data management, and are difficult to achieve as offline optimizations.

Relaxed Consistency Requirements. Maintaining data and metadata consistency in a distributed storage environment can significantly impact performance due to the overheads imposed by algorithms used to handle distributed locking and transactional namespace updates. Relaxing these consistency requirements can significantly improve application performance and scalability while reducing the overall complexity of the system, but can be challenging to integrate.

Future HPC systems will continue to increase in scale, complexity, and heterogeneity. Exploiting and adapting to current and future system diversity and complexity requires flexible application and storage system behavior. Next we describe the Legion system for developing applications that

can adapt to a wide variety of systems without requiring application changes.

2.3 Legion

Our runtime prototyping vehicle for this work is the Legion [3] runtime system, a data-centric parallel programming system for portable high-performance applications. Legion supports a logical, distributed data model that is decoupled from its implementation on memory or storage and provides us the ability to manage the consistency of application distributed state. We briefly describe elements of the Legion architecture and its support of persistent storage.

Data Model. Legion introduces and is built upon the concept of *logical regions*, an abstraction for describing structured distributed data. Logical regions are a cross product of an N-dimensional *index space* and a number of fields (a *field space*). Logical regions do not commit to and are distinct from any particular data layout or placement within memories of the machine. A logical region may have one or more *physical instances*, each of which is assigned to a particular memory with a specific layout. The data model also supports subdividing the data, either by picking out subsets of the index space or of the fields. We extend this data model to persistent storage by allowing the application data model to directly map to the underlying distributed storage model.

Memory Hierarchy. Legion models all hardware that can be used to store data as *memory*. The current Legion implementation [12] involves four kinds of memory in which instances of logical regions can be held: distributed GASNet memory accessible by all nodes, system memory on each node, GPU device memory, and zero-copy memory (system memory mapped into both CPU and GPU's address spaces). To support persistence, we introduce HDF5 and RADOS [13] memories within the Legion memory hierarchy, which allows Legion to import HDF5 files and RADOS objects into its runtime, unifying memory and persistent storage with application semantics.

Out-of-order scheduling and explicit tracking of data dependencies. Legion's data movement and computation are handled by an out-of-order scheduler that can automatically extract parallelism and identify pipelining opportunities. Coupled with a distributed data model in which dependencies between computational tasks and logical regions are explicit, the runtime can transparently overlap computation with data movement both within the traditional memory hierarchy and within the storage system. Furthermore, the runtime is able to maintain consistency of logical regions across a distributed memory through data dependency analysis among tasks accessing those logical regions.

The integration of the storage system with the Legion runtime provides opportunities to make I/O optimization decisions based on both static and dynamic system characteristics. Closer integration of the application level data model and the storage environment allows the runtime to optimize for application specific data decompositions alongside other system characteristics while insulating the application from these optimizations. These optimizations include automatic tiering, sharding, asynchronous I/O, relaxed consistency semantics, and offloading data-intensive tasks to the storage system. These opportunities are explored in the next section.

3. STORAGE INTERFACE AND RUNTIME CO-DESIGN

We propose a closer coordination between the application runtime and the lower level storage services in order to unify the application's distributed data model used in computation directly with the underlying storage model used for persistence, enabling transparent optimizations such as dynamic data placement, data distribution, and data replication based on a semantic understanding of application intent.

Figure 2 illustrates our co-design goal by showing how the runtime system with storage integration could effectively map an adaptive mesh refinement (AMR) application data structure into a distributed hierarchical storage system. To the left of the figure is a multi-resolution grid with each vertex associated with three fields (i.e., F1, F2, and F3). To the right of the figure is the integrated, distributed multi-layer storage hierarchy (for simplification, the figure only demonstrates disk and NVRAM tiers on three different nodes).

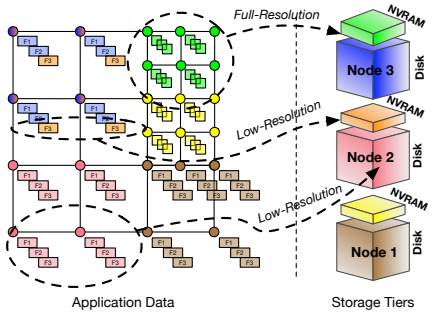


Figure 2: Legion data model and mapping to distributed and hierarchical storage.

Regions for AMR applications are often organized at different resolution levels. In Figure 2, we use green and yellow to label regions in full resolution, and use other colors for lower-resolution regions (areas with arrows marked full-resolution and low-resolution can be referred to for black-and-white print). AMR applications are often more interested in full-resolution regions, which may also come with more frequent accesses. One predominant I/O optimization for AMR application is placing full-resolution regions into a low-latency and high-bandwidth storage layer for better I/O performance. However, the resolution distribution depends on input data, which prevents this optimization from being performed at compile time. Therefore, supporting dynamic data partitioning and data placement at runtime is critical for AMR application performance.

The colors of the fields and storage tiers also represent an application-specific mapping from regions to the persistent storage hierarchy. The coloring in Figure 2 demonstrates a possible mapping decision that keeps full-resolution regions (in the top right corner of the grid), as well as field F3 for a subset of lower-resolution regions (marked in orange) in the NVRAM tier, while all other regions are stored on the disk tier. This mapping decision can be completely dynamic and made by the Legion runtime as a result of an optimization strategy, such as staging data on the NVRAM for frequent access.

Coloring. Legion employs a general method for spec-

ifying partitions of logical regions based on *colorings* that allow for arbitrary data decompositions and layout. A coloring assigns zero or more colors to each element of a logical region. Based on the coloring of each element, Legion provides a primitive partition operation that constructs subregions (partitions) of the elements of each color. As illustrated in Figure 2, the multi-resolution grid, which is itself a logical region in the Legion framework, is partitioned into subregions with different colors. The Legion runtime is able to perform different placement, re-partitioning, and layout optimizations for the different subregions.

In Figure 2 there are two colorings represented. One coloring is used to facilitate the distribution of the AMR mesh in memory, and a separate coloring on the same data structure represents the desired sharding and tiering decisions that are communicated to the storage system when the data is persisted.

Co-optimization. We believe integrating the storage hierarchy into the Legion programming model enables co-optimization opportunities between the storage layer and the runtime layer. For instance, transparent to the application, the Legion runtime can expose application specific data boundaries used to optimize data alignment and locality. Likewise, storage systems that expose details such as physical topologies, and latency and throughput estimates, can be used by Legion to schedule I/O and implement data sharding and placement strategies.

In our proposed system architecture, optimization techniques are not limited to the Legion runtime system. For instance, Legion uses an out-of-order execution model, which allows the runtime to have knowledge of pending operations. Exposing data dependencies and computational scheduling information to the storage system may allow automatic I/O scheduling among tiers, or data transformations such as re-partitioning.

4. PRELIMINARY RESULTS

We demonstrate integration of storage into the Legion memory model using a microbenchmark application that simulates a checkpoint-restart workload. The application performs a checkpoint of a globally distributed logical region into persistent storage and then reads the same data structure back into memory, approximating a restart process. We consider global checkpoint-restart because of its general use for fault-tolerance, as well as its scaling challenges in terms of I/O pattern (e.g. N-1 vs N-M), and in terms of I/O scheduling (global barrier vs. asynchronous I/O). The application initially constructs a globally distributed data structure containing the application state and provides a coloring of this state that corresponds to a desired physical partitioning on persistent storage. This coloring and the resulting distribution in persistent storage may be orthogonal to the distribution used by the computation, and the runtime handles data movement with respect to application consistency requirements. We record an I/O trace from the execution to derive the throughput of each of the read and write I/O phases, as well as to demonstrate the I/O scheduling capabilities of Legion.

While Legion is capable of the optimizations we have proposed earlier in this paper, we report only preliminary performance results in which Legion uses an application-specific coloring to control data partitioning, as well as the usage of independent I/O scheduling with respect to the data dependencies of the application expressed through the Legion programming model.

Experiment Setup. For our experiments we use two different computational environments. The first is a 308 node cluster with Intel Xeon E5-2670 processors, 32 GB RAM, Qlogic QDR IB network, and a 3.5 PB Lustre file system with 35 GB/sec peak bandwidth. The second cluster is a smaller testbed environment in which 12 nodes were configured as both RADOS clients and servers. Each node contained a modern 12-core Intel CPU, 64 GB of RAM, 512 GB SSD, and the nodes were connected using a 10 GbE network. We demonstrate the flexibility of unifying storage into the Legion runtime by running experiments on both a backend that stores HDF5 files in a Lustre file system, as well as a backend that uses raw object storage in the RADOS object storage system [13]. In the case of the HDF5 backend, the HDF5 external link facility is used to produce a top-level file that presents a unified view of the dataset through links to individual HDF5 shards.

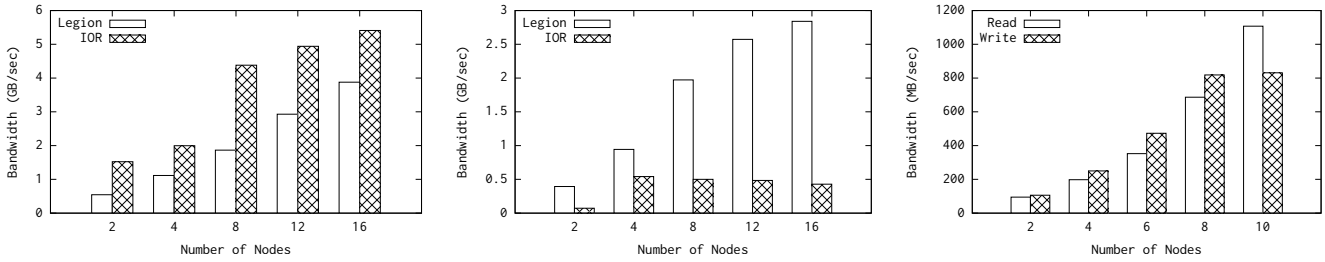
Logical region sharding maps naturally to one or more memories, and allows Legion to perform completely independent I/O while maintaining a consistent view. We relax consistency across the dataset, and require transactional consistency only at the shard level. A globally consistent view is created after shard I/O is complete using Legion's explicit tracking of I/O state.

Weak Scaling. Scaling I/O for HPC applications that perform N-1 checkpointing can be difficult when the mapping between application data and the file interact negatively with the physical alignment of the storage system. Legion can avoid this problem by transparently sharding application data and handling the complexity of managing data set consistency.

We conduct a weak scaling experiment comparing I/O performance of our Legion benchmark with IOR. We scale from 2 to 16 application nodes, using a fixed set of 256 shards, and increase total data set size from 4 GB to 32 GB. The IOR benchmark is configured to use the HDF5 API and an N-1 I/O pattern. Figure 3b plots sustained write throughput, showing that Legion I/O write bandwidth scales with the number of nodes, while IOR bandwidth begins to decrease after 4 nodes. Figure 3a shows the read bandwidth where Legion I/O scales with the number of nodes, but IOR is able to achieve a higher throughput rate in most cases due to caching affects. Figure 3c shows the same workload run against a RADOS backend. Although Figure 3c shows clear scalability, performance suffers compared to Lustre, likely due to a lack of client-side write coalescing, and a hardware configuration using a shared device for both bulk storage and write-ahead logging. However, we consider the salient point to be the flexibility of storage I/O interfaces and ability to avoid non-POSIX file systems and their short-comings at scale, such as metadata scalability, that object storage systems are less prone to suffer from.

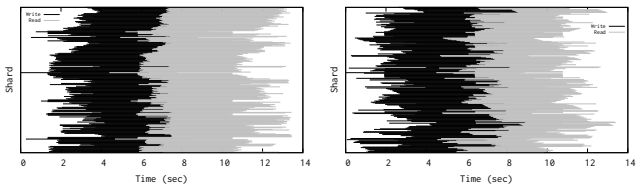
Asynchronous I/O. Checkpoint applications that use a global barrier to achieve data consistency can be inefficient when even one task is delayed reaching the synchronization point. In contrast, Legion is able to perform fully independent I/O while maintaining consistency without expensive transactions.

We demonstrate the ability of Legion to perform independent I/O by recording a trace of our microbenchmark and plot a time-series of the I/O phases (write and read) asso-



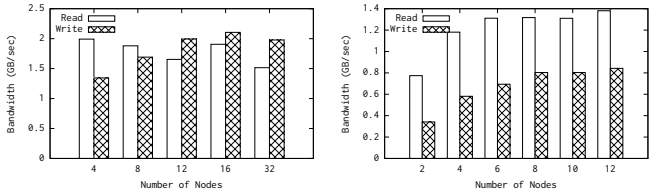
(a) Legion vs IOR (Read) (b) Legion vs IOR (Write) (c) Legion on RADOS (Read/Write)

Figure 3: Weak scaling experiment. Figures (a) and (b) show Legion I/O over Lustre compared with N-1 IOR over HDF5 over MPI-IO over Posix on Lustre. Figure (c) shows weak scaling for Legion I/O using RADOS object storage.



(a) HDF5 over Lustre (b) RADOS Object Store

Figure 4: Write and Read phases demonstrating independent I/O scheduling.



(a) HDF5 Over Lustre (b) I/O Over RADOS

Figure 5: Throughput for hard scaling experiment using 14 GB data set and 256 shards.

ciated with each shard of the global data structure being persisted. Figure 4 shows the time interval for reading and writing each of 256 shards, using both an HDF5 (Figure 4a) and RADOS (Figure 4b) storage backends. The workload was generated using 12 nodes and each shard was 56 MB for a total data set size of 14 GB. The x-axis shows time in seconds, and the y-axis represents each individual shard. The first phase (dark) is write and the second phase (grey) is read. The horizontal transition from the write phase to the read phase represents the I/O for single shard. From this experiment we can see that individual tasks within Legion may be scheduled at different times and the completion of these tasks may vary dramatically based on the performance and utilization of the underlying storage environment. Furthermore, we see that Legion is able to schedule different phases of operations based on explicit data dependencies which may allow reading tasks to run concurrently with writing tasks on the same logical region. The performance difference between HDF5 over Lustre and RADOS (HDF5 has a shorter write phase, and RADOS has a shorter read phase), could be attributed to caching. The relatively small data set size easily fits into memory on the RADOS cluster that has no other users competing for memory, compared to the shared Lustre system.

Strong Scaling. Next we present the results of a strong scaling experiment in which the problem size is held constant while we increase the number of application nodes performing I/O. This is a common scaling strategy used to reduce time-to-solution for many problem areas, and represents the I/O workload for such an application using a checkpoint-restart resilience mechanism.

The results from this experiment are shown in Figure 5 for both HDF5 over Lustre (Figure 5a) and RADOS storage backends (Figure 5b). Each graph shows the bandwidth achieved when reading and writing a total of 14 GB partitioned across 256 shards.

5. CONCLUSION

As storage systems and memory hierarchies continue to increase in complexity it has become increasingly difficult for applications to optimize I/O performance through hard-coded strategies which may involve fragile as well as non-portable data management techniques and assumptions about performance.

In this paper we consider the integration of the storage system into Legion, a next-generation application programming model and run-time that provides a hardware independent machine abstraction. We demonstrate that by understanding the semantics of application data, Legion can perform I/O optimization tasks such as data sharding completely transparent to the application, and easily supports the use of non-POSIX storage systems by handling data management tasks normally hard-coded into applications.

6. ACKNOWLEDGMENTS

This work was supported by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1, by the Advanced Simulation and Computing Program, Advanced Technology Development and Mitigation element administered by Thuc Hoang, and by Los Alamos National Laboratories Subcontract No. 173315-1 through the U.S. Department of Energy under Contract No. DE-AC52-06NA25396. Additional support came from the U.S. Department of Energy, Office of Advanced Scientific Computing Research, LA-UR-15-27390.

7. REFERENCES

[1] Introduction to hdf5. `https://www.hdfgroup.org/HDF5/doc/H5.intro.html`, 2010.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *I'12*, Los Alamitos, CA, USA, 2012.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Structure slicing: Extending logical regions with fields. In *SC '14*, New Orleans, LA, 2014.

[4] B. Behzad, S. Byna, S. M. Wild, Prabhat, and M. Snir. Improving parallel I/O autotuning with performance modeling. In *HPDC '14*, Vancouver, BC, Canada, 2014.

[5] B. Behzad, H. Vu, T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming parallel I/O complexity with auto-tuning. In *SC '13*, Denver, CO, 2013.

[6] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. Jitter-free co-processing on a prototype exascale storage stack. In *MSST '12*, Lake Arrowhead, CA, 2012.

[7] M. Houston, J.-Y. Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *PPoPP '08*, New York, NY, USA, 2008.

[8] John, , G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: A checkpoint filesystem for parallel applications. In *SC '09*, Portland, OR, 2009.

[9] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *MSST/SNAPI '12*, Pacific Grove, CA, 2012.

[10] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible I/O and integration for scientific codes through the adaptable I/O system (ADIOS). In *CLADE '08*, Boston, MA, 2008.

[11] R. Thakur, W. Gropp, and E. Lusk. On implementing mpi-io portably and with high performance. In *IOPADS '99*, number 10, New York, NY, USA, 1999.

[12] S. Treichler, M. Bauer, and A. Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *PACT '14*, New York, NY, USA, 2014.

[13] S. A. Weil, A. Leung, S. A. Brandt, and C. Maltzahn. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. In *PDSW'07*, Reno, NV, 2007.