Integrated In-System Storage Architecture for High Performance Computing

Dries Kimpe,[1] Kathryn Mohror,[2] Adam Moody,[2] Brian Van Essen,[2]
Maya Gokhale,[2] Rob Ross,[1] Bronis R. de Supinski [2]

[1] Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Argonne, IL 60439, USA

[2] Lawrence Livermore National Laboratory
Livermore, CA 94551, USA

{dkimpe,rross}@mcs.anl.gov, {kathryn,moody20,vanessen1,maya,bronis}@llnl.gov

ABSTRACT

In-system solid state storage is expected to be an important component of the I/O subsystem on the first exascale platforms, as it has the potential to reduce DRAM requirements, increase system reliability, and even out I/O load peaks.

This paper describes the design of a prototype, integrated in-system storage architecture we are developing to serve the diverse needs of high performance computing. We are developing a *container* abstraction to perform lightweight management of in-system storage devices, as well as methods to access containers remotely and transfer them within the storage hierarchy. We are also working on a storage hierarchy abstraction API to provide portable HPC I/O software with the critical information on the configuration of the system it is running on. As currently available large-scale HPC systems lack in-system storage, we are developing a solid state storage simulator backed by DRAM. These efforts are being integrated around an I/O-intensive workload provided by the scalable checkpoint/restart (SCR) library. We are hoping that once complete, our efforts with reduce the overheads of checkpointing and data movement across the system and thus improve the scalability and reliability of HPC applications.

Categories and Subject Descriptors

B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids—*Simulation*; D.4.2 [**Operating Systems**]: Storage Management—*Storage hierarchies*; D.4.5 [**Operating Systems**]: Reliability—*Checkpoint / restart*

Keywords

in-system storage, checkpoint/restart, burst buffer, I/O forwarding, storage simulation, storage management

1. INTRODUCTION

The first exascale systems, expected around 2019–2020, will enable transformative science discoveries in a number of areas, including climate, combustion, nuclear energy, and national security. A key exascale barrier is the need for scalable storage of persistent state: one that provides the necessary I/O bandwidth and capacity without overwhelming the power, cooling, and cost budgets of future systems. Traditional global storage system approaches simply cannot scale to meet these requirements.

With the development of inexpensive, nonvolatile memory technologies such as flash and phase change memory, it is feasible to include solid-state, persistent memory on every node in a future exascale system—enabling in-system storage. Such storage augments the memory hierarchy, potentially reducing DRAM requirements and thus the node's power requirements. It streamlines and simplifies checkpointing, increasing system reliability. In-system storage reduces the peak bandwidth requirements of a global exascale storage system, offering a scalable checkpoint/restart solution. However, there remain considerable research challenges to realizing these potential benefits, especially if one wants to hide from the user the complexity introduced by another layer in the storage hierarchy.

We have been conducting a detailed assessment of the potential roles and benefits of in-system storage in exascale computational science, exploring existing hardware options and assessing the software mechanisms that best exploit them. One of the early conclusions of this exploration is the identification of a need for an integrated in-system storage architecture. Different machines will employ in-system storage in different ways, and will feature devices of different capacities and capabilities. Software components will need accurate description of the configuration of the storage system; to the best of our knowledge, however, no portable API exists that would provide such information. Section 3 outlines the interfaces that we plan to provide in a form of a standalone library to fill that gap. In-system storage can serve multiple purposes, such as: (1) temporary checkpoint storage space; (2) burst buffer for application output and prefetch buffer for application input; or (3) backing store for out-of-core computations. Complex applications may well wish to employ multiple of these usage scenarios simultaneously; an abstraction layer is needed to manage the allocation of in-system storage for different purposes. Running a POSIX file system on top of a local storage device would

provide most of the necessary functionality. We believe, however, that a file system is an unnecessarily heavyweight abstraction that would introduce significant overheads, especially given the projected high performance levels of emerging solid state storage technologies. Hence, in Section 4 we introduce *containers*—a lightweight management abstraction for in-system storage. In-system storage is going to introduce new levels of hierarchy into the IO system. In addition to a global parallel file system, we can expect to see in-system storage devices local to every compute node, shared between a group of compute nodes, or attached to I/O nodes. We will need remote access to the containers and a high-performance data-transfer capability between different levels of storage hierarchy; we discuss this concept in Section 4.2. Together, these components will provide a prototype in-system storage architecture. Section 5 discusses our plans for integrating the storage architecture components with checkpoint/restart mechanisms provided by SCR and with an in-system storage simulation layer. The latter is necessary because currently accessible large-scale HPC systems lack in-system storage. Using a simulator enables us to model multiple varieties of in-system storage, including storage hardware not yet available. In addition, the simulator, yielding more detailed information than a hardware solution, simplifies optimizing our use of the storage.

2. BACKGROUND

Our current work builds on a number of existing components outlined below. Section 5 will detail how these components can be used together.

2.1 SCR

The Scalable Checkpoint/Restart (SCR) [9] library enables MPI applications to use storage distributed on system's compute nodes to attain high checkpoint and restart I/O bandwidth [7]. We derive SCR's approach from two key observations. First, a job only needs its most recent checkpoint—as soon as it writes the next checkpoint, we can discard the previous one. Second, a typical failure disables a small portion of the system, but it otherwise leaves most of the system intact. Our experiments have shown that 85% of failures disable at most one compute node on the clusters on which we currently use SCR [7].

SCR achieves high I/O bandwidth by caching checkpoints in storage local to the compute nodes instead of the parallel file system. It caches checkpoints on RAM disks, magnetic hard-drives, or SSDs, depending on what is available. SCR caches only the most recent checkpoints, discarding an older checkpoint with each newly saved one. SCR can apply a redundancy scheme to the cache, so it can recover checkpoints after a failure disables a small portion of the system. It periodically flushes a cached checkpoint to the parallel file system in order to withstand failures that disable larger portions of the system; however, a well-chosen redundancy scheme allows checkpoints to be flushed infrequently.

SCR is our primary driver for the project. We are adapting SCR to use the storage abstraction library (described in Section 3) to discover in-system storage, and to access that storage by using the container abstraction (described in Section 4).

2.2 IOFSL

The I/O Forwarding Scalability Layer (IOFSL) [1] is a portable I/O forwarding implementation. A recent addition to the I/O software stack, I/O forwarding addresses a number of different I/O difficulties typical for large parallel machines. By forwarding I/O calls to a gateway or I/O node—where they are executed locally—the number of clients visible to the file system is drastically reduced, lowering the load on the file system and enabling existing parallel file system to scale to more clients.

Another benefit of forwarding I/O is that it minimizes the overhead of I/O on the client to the bare minimum. All file system protocols are handled by the forwarding server. The client's responsibility is limited to the communication needed to transfer the I/O data to and from the forwarding server. This is particularly beneficial for systems with a large number of relatively less powerful compute nodes, and is especially relevant for systems using a microkernel on the compute nodes, as these kernels often lack the functionality needed to implement a full I/O stack.

We are extending IOFSL to allow for forwarding of API calls for in-system storage (further described in Section 4.2), leveraging its existing networking and forwarding code. In addition, IOFSL will be responsible for coordinating and moving data between the different storage levels (node, gateway or I/O node, parallel file system) in the system.

2.3 PerMA

The PerMA simulator [11] is being developed to allow us to model the impact of future generations of I/O-bus-attached NVRAM on application performance at scale, and prototype the impact of a high performance memory-map runtime. The simulator provides a memory API to the persistent memory, models latencies and bandwidths ranging from current Flash down to DRAM-like performance, and supports hundreds of threads at native speed.

The simulator has provided new insights into the interaction of algorithmic techniques (e.g., thread oversubscription or out-of-core data structures) with future NVRAM technology generations. Furthermore, the work on the PerMA simulator illustrates that it may be possible to get scalable performance for out-of-core algorithms, with a high performance memory-map runtime and the appropriate caching algorithms. Our exploration of the current memory-map runtime within Linux (in [11]) has identified several bottlenecks for high performance computing, and has lead to the initial development of the data-intensive memory-map runtime (DI-MMAP) within the PerMA simulator.

We have implemented a simple, two-level buffering scheme for DI-MMAP within the simulator to provide locality-optimized page mapping. We have shown that the combination of a latency-tolerant, concurrent algorithm, future NVRAM devices, and an optimized memory-map runtime system enables migration of data structures that were traditionally heap-allocated into persistent memory.

3. STORAGE HIERARCHY ABSTRACTION

The goal of the storage hierarchy abstraction is to provide a generic interface for other components to discover available storage resources and their properties in a scalable and portable way. We strive for generality given the highly-specialized and diverse hardware and software on supercomputers. This will enable tools that use the interface to be portable across a wide variety of machines, assuming

the storage hierarchy API has been ported to a particular machine. Scalability is important on today's machines and will be critical on future systems. Because of this, we designed the API so that the memory requirements will not grow with the number of components in the machine.

Each storage location has a Unique Resource Identifier (URI) as well as a set of properties. URIs are strings that identify a unique store and can be used to locate it. As an example, a storage URI might be `storage://compute2/tmp`, which would refer to the storage location `/tmp` on compute node `compute2`. For flexibility, properties are attribute-value pairs. We could have opted to define a set of properties for each store, but this would require an API change whenever additional properties were needed for a new storage device or client application with different requirements. Instead, the interface will return an array of attribute-value pairs for each store, the values of which will be documented for a given system. For example, an attribute could be `capacity` with value `10E12`.

In order to be scalable, it is important that a single component instance need not store the entire map of the storage hierarchy. Our approach is that components only need to query for storage resources as needed. A component can query the storage locations local to it as well as storage up and down the hierarchy. For example, a component running on an I/O node on a cluster might query the storage hierarchy abstraction for the location of the next level of storage in the hierarchy, e.g., the parallel file system or some other large, intermediate storage device. After receiving a list of the storage device(s) at the next level, the component can query the API to discover the properties of a store to see if it meets the needed requirements. For example, it can query to find out if `capacity` is adequate to store the data required. If not, it can query for the levels of storage further down the hierarchy on an as needed basis.

4. CONTAINER ABSTRACTION

Analyzing the in-system storage requirements of applications (out-of-core techniques), checkpointing and I/O system libraries, it is clear that providing a full fledged "file" abstraction for in-system storage is not required. Furthermore, supporting unused features such as global immediate consistency, sparse files, arbitrarily deeply nested subdirectories, and on-demand resizing incurs a high implementation and run-time cost. At the same time, the specialized environment and use case of in-system storage benefits from features not commonly found in general purpose file systems. Retrofitting these on top of an existing file concept is often not feasible. An example of such a feature is zero-copy application I/O, made possible by leveraging the ability of most in-system storage to be mapped directly into the node's memory hierarchy.

Instead, a new model, specifically designed for managing and accessing in-system store, is proposed. Designing and implementing a special-purpose model has a number of additional benefits, such as the flexibility to investigate non-traditional I/O semantics and APIs. In addition, a library, user-space only implementation simplifies deployment (for example on top of the PerMA simulator described in Section 2.3), as no administrator access (which is often not available) is required. To differentiate our implementation from the traditional file concept, the basic storage entity of our proposed in-system storage model is referred to as a *container*.

4.1 Local Container Access

While a container resembles a POSIX file, there are a number of significant differences. Containers are created with a specific size. Writing past the end of a container is not allowed, and will not implicitly resize the container. However, it is possible to explicitly resize containers, but doing so is a high-cost operation. While this might seem overly restrictive, it significantly simplifies the implementation and run-time performance of the container. In particular, it enables providing direct memory access to the underlying storage forming the container.

As parallel applications often exhibit highly complex I/O patterns [3], the container API provides for fully non-contiguous container access, both for reading and writing. This means it is possible to transfer a non-contiguous set of bytes from the container to a non-contiguous set of bytes in the application's memory, or vice-versa.

Due to cost, manufacturing, power, and space reasons, most deployed and planned in-system storage is provided by solid state devices such as flash or phase change memory. It is possible to expose these devices to software by mapping them into the system's memory address space. This makes it possible for the system to perform zero-copy I/O by directly generating or accessing data into the device-mapped memory range. Where supported by the system, the container API preserves the ability to perform zero-copy I/O by allowing an application to retrieve the mapping between the logical and on-device layout of a container. In most cases, we predict that using direct access to the container enables a better performing and less complex I/O mechanism than performing explicit read or write calls.

Each container has a name—a set of characters—provided at create time. Aliasing (for example through links) is not supported. Containers only support a restricted set of operations: create, delete, rename, resize, get attributes, read, and write.

Instead of providing full directory semantics, containers are grouped within a *container set*. Container sets provide isolation from other software layers (within the same application or from other jobs), as each container set represents a different namespace. In addition, container sets support *space reservation*, simplifying application usage by alleviating the need for the handling of out of space conditions. By enabling reserving space ahead of time, storage fragmentation of containers and sets can be kept to a miniumum. On systems requiring data access protection, access control can be implemented on the container set level; for example, container sets can be protected using a cryptographic secret, provided by the user at job submission time. The API supports listing all of the containers in a set.

4.2 Distributed Container Access

While the container model is a purely local abstraction, remote container access can be advantageous in certain situations. For example, SCR might need to access the storage located on the I/O or gateway nodes, even though those nodes typically don't support running user code. To enable this and other use cases, a remote container API enables access to containers on locations other than the local node. The API leverages the URI abstraction provided by the stor-

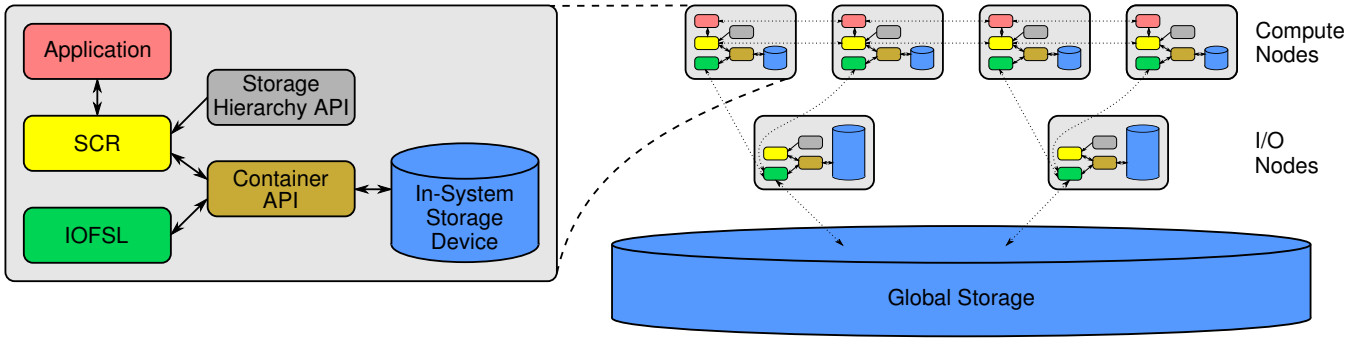

Figure 1: The prototype, integrated in-system storage architecture. Dotted lines indicate communicating components.

age hierarchy abstraction (see Section 3) to identify remote containers.

It is important to note that this approach does not create a global name space. It merely offers an abstraction for shipping an operation and associated data to the remote container storage, where the local container API is used to execute the requested operation.

In addition to enabling remote access, the extended API also provides for third-party container transfer. This means that a node can request for a container to be copied or moved between two *other* nodes, with both source and destination distinct from the requesting node. This functionality can be used by SCR to incrementally drain certain checkpoints from the on-node storage to the parallel file system.

4.3 Implementation

The local container API is provided by a C/C++ library. Our current implementation supports multiple storage backends. To simplify testing, one backend supports allocating main memory to serve as the in-system storage. However, the default backend assumes in-system storage can be mapped into main memory using the mmap system call. This model is supported by the PerMa simulator, enabling easy integration between the two components.

All provided functions are fully thread-safe, and can be called concurrently by threads within the same application (process) or concurrently by multiple different applications. This is needed as both SCR and IOFSL might attempt to access the same container concurrently.

As an example, the declaration of the cs_container_read function is shown below (the corresponding cs_container_write has the same prototype, except for a difference in the constness of the memory parameter).

```
int cs_container_read (cs_container_handle_t handle,
    size_t memcount, void * membuf[ ],
    size_t memsizes[ ], size_t filecount,
    cs_off_t fileofs[ ], cs_off_t filesizes[ ],
    cs_off_t * transferred);
```

The cs_off_t aliases to an appropriate integer type, guaranteed to be at least 64 bits wide. Both destination and source can be described as a series of contiguous byte ranges, enabling scatter/gather functionality. The handle parameter, obtained by a call to cs_container_open, identifies the container to be accessed.

The zero-copy access functions, cs_container_zc_read and cs_container_zc_write, are very similar to the corresponding cs_container_read and cs_container_write functions, except for the fact that the former do not specify where the retrieved data has to go. Instead, the output of these functions is a list of memory ranges. Each of these ranges refers to the memory mapping the in-system storage. The underlying container can then be accessed by reading or writing these memory ranges. A mapping remains valid until the underlying container is removed or resized, enabling very fast access to the in-system storage. This is especially useful functionality for the SCR redundancy schemes, as the computed data can be directly stored in the container without first having to go through an intermediate buffer. In addition, the zero-copy access functions offer excellent support for applications using an out-of-core strategy.

Except for an extra argument indicating which node to contact, the remote container API is very similar to the local container API. Under the covers, the remote container API is implemented by IOFSL. IOFSL takes care of forwarding the API call to the destination node, executes the call at the destination using the local container API, and sends the output of the call back to the requesting node.

Since the zero-copy API requires the ability to directly map the storage device into the application's memory space, this functionality is limited to processes co-located with the storage. Therefore, access to remote containers is restricted to traditional read and write functions, necessarily involving a copy from the application's buffer to the in-system storage.

Container transfers are also handled by IOFSL. Since IOFSL instances can communicate with one another, transfers can be scheduled appropriately to optimize bandwidth usage. Data can be moved between two containers or between a container and the parallel filesystem. Once a transfer is initiated, control returns to the application while IOFSL takes care of moving the remaining data in the background.

5. INTEGRATION

Figure 1 provides an overview of the integrated system architecture we are working on. The figure shows a system having two node types: compute nodes and I/O nodes. Compute nodes make up the main part of the machine, while a smaller number of I/O nodes provides system services such as I/O and node monitoring. Many of the contemperorary large HPC machines, for example the IBM Blue Gene series [5], follow this architecture.

SCR currently supports writing a checkpoint to a file and, as indicated in Section 2.1, it supports a variety of local storage devices to cache checkpoints, including solid state drives, so long as there is a file system running on top. Our

container abstraction from Section 4 does not implement a file abstraction, though we are planning to provide a limited emulation layer to ensure portability of existing checkpoint/restart codes. Long term, however, we are assuming that the best checkpointing interface will be through a high performance memory-map runtime such as DI-MMAP. Specifically, the checkpoint container will be mapped into the address space of an application process and the data to be checkpointed will simply be transferred using memory copy operations. Irrespective of the exact mechanism used, SCR will ensure that the data ends up in a container allocated on a local in-system storage device.

Container API is local, whereas SCR needs to be able to transfer data between compute nodes to provide redundancy of cached checkpoint files and periodically also needs to flush the latest checkpoint to the parallel file system. We will provide an infrastructure to copy containers between different levels of the storage hierarchy as described in Section 4.2, based on our earlier I/O forwarding project IOFSL. The advantage of offloading that functionality from SCR to a separate component is that the transfers can then be performed asynchronously and can be coordinated across multiple nodes based on system load, the relative importance of the data to be transferred, etc.

We will support container transfers between different levels of the storage hierarchy, but for now not within a single level, at least not between different compute nodes. Contemporary large-scale HPC systems such as IBM Blue Gene/Q or Cray XE6 simply do not offer convenient interfaces to perform such transfers independently of the application. Instead, we will use existing, tried and tested SCR mechanisms of performing such transfers at application-level using MPI primitives; hence the horizontal arrows between SCR components in Figure 1. Please note that while the figure shows individual components on the compute nodes as independent entities, in practice most of them will be linked together into the application process.

As existing large-scale HPC systems do not have in-system storage, we will simulate such storage devices in DRAM, using the PerMA simulator as indicated in Section 2.3. The current prototype implementation of the container library expects the storage to be mapped into a contiguous memory region, which is an interface well supported by the simulator. We plan to evaluate various scenarios, such as local storage on every compute node, storage attached to I/O nodes, or various combinations and intermediate options. This will also give us an opportunity to solidify the storage hierarchy abstraction API, which SCR will depend on to obtain system configuration information. We expect larger in-system storage devices above the level of individual compute nodes, such as on I/O nodes as shown in Figure 1. We plan to put additional SCR processing components at this level, which will be performing comparisons of checkpoints from different compute nodes, eliminating inter-node redundancies.

6. RELATED WORK

Several efforts provide abstractions to hierarchical relationships between components on systems. The Platform Description Language (PDL) is a generic, XML-based language for describing the relationships between compute components on heterogeneous systems [8]. Using PDL, program tasks can be mapped to appropriate computational units based on their characteristics, including memory regions and interconnects. PDL differs from our approach in that it focuses only on components within a node and does not address outside resources. Sequoia is a programming model for abstractly describing hierarchies of tasks and mapping them to the memory system of a target machine [4]. While Sequoia is a portable approach to describing system components, it focuses solely on the memory hierarchy. The Portable Hardware Locality (hwloc) software provides a portable abstraction for discovering the hierarchical topology of system components [2]. hwloc typically provides information about processing units and memory but can also discover I/O devices. However, while hwloc can discover I/O devices physically located on the host, our storage hierarchy API can provide information about storage locations available from the current host as well as from other hosts on the system.

Multiple distributed file systems have been described in the literature, each of which was designed for high performance with a particular workload in mind. The Google File System (GFS) [6] and the Hadoop File System (HDFS) [10] were designed for a workload that typically consists of large files, with append-only writes from concurrent writers, on commodity clusters where failures are common. GFS and HDFS replicate files for high availability. They do not implement a POSIX interface, but instead support only a subset of the usual operations. The Ceph file system was designed for large-scale jobs with potentially hundreds of thousands of concurrent I/O requests [12]. Ceph also supports data replication for reliability and a near-POSIX interface; it relaxes consistency semantics for read-write sharing when possible. Like GFS, HDFS, and Ceph, our container API is designed to benefit a particular workload and does not support full POSIX semantics. However, our workload is considerably different from those described above. To the best of our knowledge, ours is the first storage management system designed to improve the utilization of hierarchical in-system storage on high performance computing systems.

7. CONCLUSION

The DOE NoLoSS project conduct detailed assessment of the potential roles and benefits of emerging storage technologies and I/O architectures in exascale computational science. In this paper we have provided an overview of a prototype, integrated in-system storage architecture that we are developing. It includes a checkpoint/restart library (SCR), I/O forwarding layer (IOFSL), solid state storage simulator (PerMA), as well as new abstraction layers being developed to tie these components together: the storage hierarchy API and the container API.

The software we are developing is open source. We particularly encourage the reuse of the general-purpose libraries such as the container abstraction or the storage hierarchy abstraction; these will be released as independent components once they reach the necessary level of maturity.

The integration work is ongoing, and the outcomes will be the subject of future publications. We hope that our findings will influence the designs of future solid state storage devices, the I/O architectures of future extreme-scale systems, and the exascale I/O system software.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory.
This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, 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 article or allow others to do so, for United States Government purposes. LLNL-CONF-557032-DRAFT.

8. REFERENCES

[1] Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., and Sadayappan, P. Scalable I/O forwarding framework for high-performance computing systems. In *Proceedings of the 11th IEEE International Conference on Cluster Computing (CLUSTER'09)* (Tsukuba, Japan, Sept. 2009).

[2] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'10)* (Pisa, Italy, Feb. 2010), pp. 180–186.

[3] Crandall, P., Aydt, R., Chien, A., and Reed, D. Input/output characteristics of scalable parallel applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing (SC'95)* (San Diego, CA, Nov. 1995).

[4] Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., Park, J. Y., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. Sequoia: Programming the memory hierarchy. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'06)* (Tampa, FL, Nov. 2006).

[5] Gara, A., Blumrich, M., Chen, D., Chiu, G., Coteus, P., Giampapa, M., Haring, R., Heidelberger, P., Hoenicke, D., Kopcsay, G., et al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development 49*, 2/3 (2005), 195–212.

[6] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. *SIGOPS Operating System Review 37*, 5 (Dec. 2003), 29–43.

[7] Moody, A., Bronevetsky, G., Mohror, K., and de Supinski, B. R. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)* (New Orleans, LA, Nov. 2010).

[8] Sandrieser, M., Benkner, S., and Pllana, S. Explicit platform descriptions for heterogeneous many-core architectures. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)* (Anchorage, AK, May 2011), pp. 1292–1299.

[9] Scalable Checkpoint/Restart Library. `http://sourceforge.net/projects/scalablecr/`.

[10] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)* (Lake Tahoe, NV, May 2010).

[11] Van Essen, B., Pearce, R., Ames, S., and Gokhale, M. On the role of NVRAM in data-intensive architectures: An evaluation. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS'12)* (Shanghai, China, May 2012).

[12] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)* (Seattle, WA, Nov. 2006), pp. 307–320.

Government License