Mediating Data Center Storage Diversity in HPC Applications with FAODEL

Patrick Widener[1(✉)], Craig Ulmer[2], Scott Levy[1], Todd Kordenbrock[3], and Gary Templet[3]

[1] Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, USA
patrick.widener@sandia.gov
[2] Scalable Modeling and Analysis Systems, Sandia National Laboratories, Livermore, CA, USA
[3] Perspecta, Inc., Chantilly, VA, USA

Abstract. Composition of computational science applications into both *ad hoc* pipelines for analysis of collected or generated data and into well-defined and repeatable workflows is becoming increasingly popular. Meanwhile, dedicated high performance computing storage environments are rapidly becoming more diverse, with both significant amounts of non-volatile memory storage and mature parallel file systems available. At the same time, computational science codes are being coupled to data analysis tools which are not filesystem-oriented. In this paper, we describe how the FAODEL data management service can expose different available data storage options and mediate among them in both application- and FAODEL-directed ways. These capabilities allow applications to exploit their knowledge of the different types of data they may exchange during a workflow execution, and also provide FAODEL with mechanisms to proactively tune data storage behavior when appropriate. We describe the implementation of these capabilities in FAODEL and how they are used by applications, and present preliminary performance results demonstrating the potential benefits of our approach.

Keywords: Workflow · Composition · Data management · Scalability

1 Introduction

Traditionally, I/O for data storage in high-performance computing applications (especially computational science simulations) has almost always meant data transfer from node DRAM to a parallel file system (PFS) such as Lustre or GPFS. That traditional arrangement has been destabilized in a number of ways:

– Impedance mismatches, between the rates at which application data is generated (through simulation of physical phenomena or capture from external sources) and the available bandwidth to stable storage provided by datacenter-scale PFS, have not abated.

SAND2019-6668C.

M. Weiland et al. (Eds.): ISC 2019 Workshops, LNCS 11887, pp. 275–287, 2019.
https://doi.org/10.1007/978-3-030-34356-9_22

- Available PFS solutions in many cases require application-specific configuration and tuning, and continue to be a major source of resilience issues.
- Partly as a response to the above points, the storage hierarchy continues to grow deeper and more complex. Potential layers include local and remote memory (e.g., on package high bandwidth memory, DRAM, nonvolatile memory (NVM), 3D-stacked DRAM); compute area Storage Class Memories such as burst buffers; parallel file systems; campaign storage; and archival storage. Each level in the hierarchy has its own operational tradeoffs that users must understand to efficiently leverage the underlying storage resources in their application.
- The influence of data movement between host and accelerator memories, and the desire to maintain zero-copy performance as that data must be moved to stable storage, has increased the influence of APIs such as OpenACC [1] and Kokkos [7].

Additionally, the abstractions commonly available to application developers will not support the development and deployment of future exascale systems. Several important trends in application design and deployment are highly dependent on the availability of high-performance and semantically-flexible I/O services:

- *Coupled Simulation Codes:* Increasingly, important scientific simulations require multiple physical models to be evaluated simultaneously. To leverage existing software, one way to combine these physical models is to run each model independently and map the output of one physical model to the input for another, and vice-versa. In this way, physical models that are captured by independent executables exchange information about the state of the simulated system as the simulation progresses. Achieving high performance for these coupled simulation codes requires the availability of services that facilitate the efficient exchange of data between multiple physical models. Moreover, programmer efficiency is dependent on simple and robust mechanisms for exchanging data between coupled simulation codes.
- *Complex Workflows:* Analysis of scientific simulation data commonly requires processing by a sequence of several specialized analysis tools; the output of one is the input for the next. The analysis tools that comprise these workflows include: mesh generation and mesh refinement tools, preconditioners, uncertainty quantification tools, simulation frameworks, solvers, and visualization/analysis tools. Workflow management tools have typically exchanged data via a parallel filesystem. However, the PFS I/O bandwidth limitations throttle severely the amount of work that can be done, while the gap between compute speed and IO bandwidth continues to increase.
- *Asynchronous Many-Task (AMT) Programming Models:* AMT programming models (e.g., Legion [4], Charm++ [14], and Uintah [10]) are designed to allow compilers and the associated AMT runtimes to manage the complexities that arise due to performance variation and resource heterogeneity [18]. Moreover, the asynchronous nature of these models allows them to overcome many of the

performance costs that are borne by bulk synchronous parallel (BSP) codes on extreme-scale systems. However, these characteristics of AMT applications mean that predicting where a task may execute is not straightforward. As a result, AMT runtimes have commonly relied on the parallel filesystem to facilitate access to applications. Because of the costs associated with parallel filesystem access, the performance of AMT-based applications will be highly dependent on the ability of AMT runtimes to leverage the entire storage hierarchy to provide AMT tasks with efficient access to application variables.
- *Beyond POSIX storage:* Many widely used tools from the high-performance data analytics (HPDA) space are oriented toward data storage without using traditional file system interfaces. Apache's Spark [9] toolset is an exemplar of this approach. Spark relies on data access capabilities provided by the non-POSIX APIs of sources like HDFS, Cassandra, and others which may themselves rely on traditional filesystems to varying extents but do not expose those interfaces to their users. As HPDA becomes a more popular component of workflows, sharing data in a single data center with HPDA tools means finding ways to coexist with their data management strategies.
- *Resilience:* The dominant approach to fault tolerance is checkpoint/restart. Minimizing the performance impact of checkpoint/restart requires services that provide efficient access to persistent storage resources. Moreover, while checkpoints have traditionally been stored in parallel filesystems, techniques for leveraging the entire storage hierarchy have begun to grow in importance, *cf.* [16].

FAODEL [19] provides a set of data movement, storage, and management services designed to address these challenges for next-generation HPC applications and workflows. FAODEL's advantages include:

- *Programmer efficiency:* presenting application programmers with a single interface for data movement lowers the burden on application programmers, reduces development costs, and lowers the risk of mistakes as programmers attempt to master multiple interfaces to data movement services.
- *Shared optimization:* because FAODEL provides data movement services through a unified interface, optimization and validation of shared components can provide performance benefits in multiple data movement scenarios.
- *Aggregated storage resources:* because FAODEL provides a unified interface to multiple levels of the storage hierarchy, it can dynamically make decisions that allow it to avoid storage resources that are slow or have high energy costs unless absolutely necessary (e.g., avoiding the parallel filesystem in favor of node-local storage: DRAM, SSDs, NVRAM).

In this paper, we describe recent work which expands upon the last of these advantages. Specifically, we describe how FAODEL allows applications to choose, in a semantically appropriate manner, different persistent storage destinations for different subsets of the data they produce or exchange with other applications. These decisions are typically driven by a tradeoff space that encompasses the available storage hardware, the locality properties of the data in question, and

whether other tools requiring specific data management are being used. This kind of tradeoff space is already not uncommon: data center managers seek to leverage existing power and cooling installations; same-platform deployment of both HPC and HPDA applications is becoming a priority; and as stated above increasingly diverse storage hierarchies are now in wide deployment.

Our discussion is structured as follows. Section 2 briefly recaps the structure of FAODEL, with emphasis on the Kelpie service within which our work described here is implemented. Section 3 describes our implementation of mediated storage within Kelpie. Sections 4 and 5, discuss related work and conclude our discussion, respectively.

2 FAODEL Background

We briefly discuss relevant components of the FAODEL service in this section. An overview of the relationships between the software components that comprise FAODEL is shown in Fig. 1. A more detailed description of other FAODEL components can be found in [20]. High-level components that are most relevant to application developers in the remainder of this section.

2.1 Kelpie

Kelpie provides a *key/blob* abstraction to facilitate flexible data exchange between different executables (e.g., a simulation application and applications

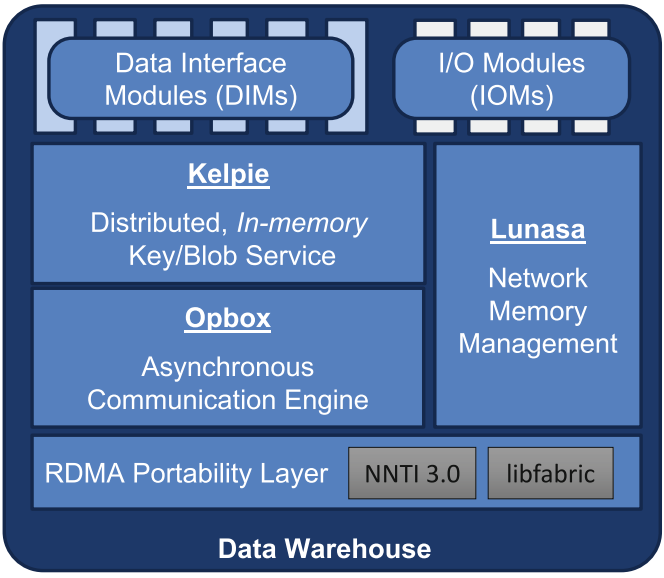

Fig. 1. Software architecture overview of FAODEL.

for visualization and analysis). A *key* is a programmer-defined text string that allows the programmer to attach semantic significance to the associated data: a *blob*. Although a key may attach programmer-cognizable meaning (and possibly structural information) to a blob, Kelpie is entirely ignorant of any meaning attached to keys or blobs.

Independent processes can exchange data in Kelpie by simply exchanging keys. The semantics of the keys exchanged may be *implicit*, the processes involved in the exchanges are unaware of the keys' semantics, or *explicit*, the processes involved in the exchange can extract meaning from the key. For example, a programmer may construct a key by encoding metadata (e.g., the application name, run number, iteration number, and variable name) that describes the contents of the associated blob. Based on shared knowledge of the key's encoding, the recipient of a key can extract the metadata from the key to inform its handling of the blob.

A key abstraction in Kelpie is represented by `Pool` objects. Each `Pool` object represents a collection of resources (e.g., nodes) that support a key/blob store. A `Pool` supports three basic operations: *Publish*, *Want*, and *Need*. *Publish* allows the user to add a key/blob pair to the `Pool`. *Want* and *Need* allow the user to request the blob associated with a key in the `Pool`. The distinction between the two is that *Want* is a non-blocking operation and *Need* is a blocking operation.

2.2 I/O Management (IOM) Modules

One of the services provided by Kelpie is to allow users to request the transfer of key/blob data to persistent storage. The interface between Kelpie and persistent storage resources (e.g., NVRAM, parallel filesystem, databases) is managed by I/O Management (IOM) modules. IOMs are built on high-level APIs (e.g., POSIX-compliant filesystems, HDF5, LevelDB) that provide access to the underlying storage resources. Each `Pool` is associated with an IOM that provides access to a particular storage resource interface (e.g., POSIX, HDF5).

FAODEL provides applications with services for transferring data to storage resources throughout the system's storage hierarchy. Each tier in the storage hierarchy provides different access characteristics that are leveraged by different use cases.

Various types of storage resources are accessible through Kelpie's IOM modules:

- **Distributed memory.** Distributed memory provides access to the collective DRAM (conventional DRAM devices and 3D-stacked DRAM devices) within the application's hardware allocation. Relative to other storage resources, distributed memory provides low-latency, high-bandwidth storage. RDMA transfers allow for efficient access to remote memory resources. Distributed memory can be used by AMT runtimes to store and exchange application variables and by coupled codes to exchange simulation data.
- **Local persistent storage.** Local persistent storage resources include SSDs and NVRAM. Locality varies by system. In some cases, persistent storage may

be available on each compute node, other systems may provide per-chassis or per-rack persistent storage resources. Local persistent resources can be leveraged as part of a checkpoint/restart solution (*cf.* [16]). Similarly, because these devices typically provide much more storage capacity than volatile memory (i.e., DRAM), they may also be used in support of *in situ* analytics.
- **Burst buffers.** Recent HPC systems such as the Cray XC40 (deployed at Los Alamos National Laboratory and the National Energy Research Scientific Computing Center) and the IBM CORAL system provide fast non-volatile storage colloquially referred to as *burst buffers*. These resources are made available to compute nodes via vendor-specific libraries (e.g., Cray DataWarp) and integrated via high-speed interconnects.
- **Archival storage.** In most systems, the principal archival storage resources are provided by a parallel filesystem. Archival storage provides high-latency, low-bandwidth access to high-capacity storage devices (e.g., hard disks).

3 Mediating Storage Using Kelpie Object Naming

Applications use the Kelpie interface to specify data they wish to store, retrieve, or exchange with other applications using the service. Like in other key-value stores, applications can use Kelpie's key structure to represent a namespace whose components have semantics appropriate to those applications and to application-to-application interactions. In this way, the namespace can convey important information about the data being exchanged and help developers reason about the structure of the problem being addressed.

Our work here explores the use of the Kelpie namespace to reflect information about how Kelpie handles data storage. A common conceptual distinction in computational workflows is the notion of a *control plane* of metadata about the current problem being solved and a *data plane* of result data from simulation or analysis. The difference between these two is the amount of data and how it is used. The control plane typically comprises larger numbers of smaller data items which are more frequently used. This use case is well supported by storage on solid-state media where reads and random access are advantaged. Volume data comprising smaller numbers of larger data sets, conversely, is better suited to bulk parallel file systems which are optimized for this case. An orthogonal case which also can be addressed here is when certain data must be shared with other applications that do not rely on file system interfaces, instead using byte-addressable interfaces or relying on services such as NoSQL databases.

We describe in this section how we support the annotation of Kelpie object namespaces with enough information for Kelpie to perform *storage mediation*. In this way, determination of persistent storage destinations for data can be based (to varying degree) on how that data is named. This provides multiple benefits. Applications can structure the namespace to hint to Kelpie about the relative "shape" of their data (signaling metadata vs. result data, for example). A partition of the namespace can be dedicated to storage via non-POSIX methods, allowing other workflow components to better understand which data is being

produced for which purposes. Also, our approach provides mechanisms for Kelpie to either cooperate with application-structured namespaces (and the implied storage hints), weigh those hints alongside internal considerations which need not be exposed to applications, or restructure or even ignore application namespace partitioning entirely.

3.1 Kelpie Architectural Considerations

Kelpie Namespaces. Kelpie implements different key indices; for the purposes of our discussion we concern ourselves with its distributed hash table (DHT) implementation (its details are similar to implementations in other KV stores). Kelpie's API provides either a one- or two-dimensional namespace. In practice, this allows applications to easily separate 2-dimensional data (row vs. column) for efficient distributed indexing. A Kelpie key can be anything serializable to a string. For a one-dimensional Kelpie keyspace, a hierarchical tree-based name structure (similar to that used in POSIX file systems) can be defined. This is the type of namespace we consider in this work.

Kelpie Persistent Storage. FAODEL (of which Kelpie is a component) is designed as a memory-to-memory data management system, where running Kelpie instances on separate nodes cooperate in DHTs by storing and providing data in node memory. However, Kelpie also supports persistent storage of data to satisfy resilience requirements or to relieve pressure on node memory allocations. This persistent storage is managed by Kelpie's *I/O management* (IOM) subsystem. Each Kelpie instance has associated with it an IOM object which provides access to a particular kind of persistent storage. IOM types include file system storage supported by POSIX and HDF5 APIs, lightweight KV storage implemented in LevelDB [11,12], and the Apache Cassandra column-oriented database [8,15].

3.2 Annotating the Kelpie Namespace

Applications interact with Kelpie through a `Pool` object, issuing *Want*, *Need*, and *Publish* operations for data objects located at given points in the namespace managed by the `Pool`. We did not want to change this interaction for existing Kelpie clients, so we added an aggregation object called `Metapool`. A `Metapool` mimics the interface of a `Pool`, allowing clients to use it in the same manner. Calls to this interface are delegated to a collection of `Pools` which are managed by the `Metapool` (Fig. 2). A newly created `Metapool` cannot be used until it this collection of `Pools` is provided.

Applications using a `Metapool` acquire `Pool` objects in the normal manner. Each `Pool` object is registered with the `Metapool` along with a C++ function closure or *lambda* whose function signature is `bool fn(const std::string& keystr)`. Each call to the `Metapool` object's *Publish*, *Want*, or *Need* methods takes the key string (supplied as a required parameter) and searches through the

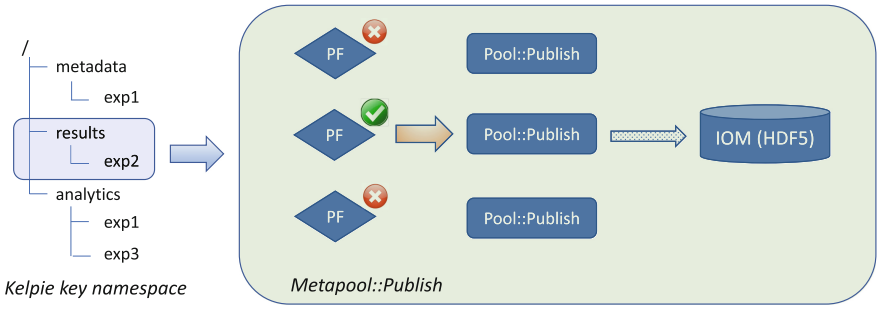

Fig. 2. A *Publish* operation makes data available at a particular location in the Kelpie namespace. Using the `Metapool` object, the location is examined to determine which of the managed `Pools` should handle the request. Each `Pool` can be configured with a different persistent storage strategy through its IOM component, giving applications a means of selecting their preferred storage approach. In a hypothetical example depicted here, the application makes large experimental result data available at */results/exp2*, and its `Metapool` has been configured to delegate management for data under */results* to a `Pool` whose persistent storage method is the HDF5 library.

collection of `Pools` in order of their registration, calling the associated function closure for each registered `Pool`. The first function closure called that returns `true` indicates that its paired `Pool` is the one which should be delegated this call from the `Metapool` object. This results in, for example, a *Publish* operation being delegated to a particular `Pool`, which has an IOM subsystem targeted at a particular kind of storage (HDF5 vs POSIX vs LevelDB, etc.).

This arrangement gives application developers a great deal of flexibility in partitioning the namespace. For example, assume two `Pools` are in use, P1 and P2. P1 is registered with the `Metapool` using a function closure that returns `true` if the given key string has a prefix of `/metadata'` and is configured to store data persistently using LevelDB. In similar fashion, P2 is registered using a function closure returning `true` for keys prefixed by `/results`, and is configured to store data persistently using HDF5 (Fig. 3). Under this arrangement, the application can store control plane information using the `/metadata` key prefix and exploit lightweight storage for that data. This would of course depend as well on how LevelDB was configured, and in this scenario configuring LevelDB to use locally accessible NVRAM would be appropriate. The end effect is that of a single namespace available through and managed by the `Metapool` object, which after configuration provides data persistence to different storage targets without any additional intervention by the application (Fig. 4).

3.3 Service-Initiated Mediation

The `Metapool` implementation also provides Kelpie with the means of mediating the configuration of `Pools` requested by an application. Since Kelpie manages the collection of `Pools`, it can introduce changes to `Metapool` handing at run

```
hdf5_dht = kelpie::Connect( "ref:/myapp/results" );
leveldb_dht = kelpie::Connect( "ref:/myapp/metadata" );
cassandra_dht = kelpie::Connect( "ref:/myapp/analytics" );

metapool.Manage( hdf5_dht,
[]( const kelpie::Key& k ) {
  if( k.K1.size() < 9 ) return false;
  if( k.K1.substr( 0, 8 ) != "/results" ) return false;
  return true;
} );

metapool.Manage( leveldb_dht,
[]( const kelpie::Key& k ) {
  if( k.K1.size() < 10 ) return false;
  if( k.K1.substr( 0, 9 ) != "/metadata" ) return false;
  return true;
} );

metapool.Manage( cassandra_dht,
[]( const kelpie::Key& k ) {
  if( k.K1.size() < 11 ) return false;
  if( k.K1.substr( 0, 10 ) != "/analytics" ) return false;
  return true;
} );
```

Fig. 3. An example of configuring the `Metapool` object from client code. The application supplies lambda functions to the `Metapool` object through the `Manage` method, associating each with a particular Kelpie `Pool`.

time, in response to changing system or workflow conditions. Different strategies that Kelpie can employ for such service-initiated mediation include:

- *Weighting partition function responses.* If the local Kelpie configuration has enough information about local storage configuration, it might be appropriate to assign different weights to the filter functions registered with the `Metapool` (as opposed to the nominal situation where the first `true` response is considered 100% authoritative. This could prove useful in a case where a Kelpie application configured for one data storage environment is ported to a different environment.
- *Changing IOM configuration at runtime.* Different storage targets can be assigned as a workflow execution evolves, and additional targets might be added as a form of load balancing.
- *Disregarding partition function responses entirely.* At times it may make sense to disregard the application's suggested namespace partitioning entirely and route data to a specific storage configuration. This also might be useful in the case of porting a Kelpie application to a new storage environment.

3.4 Performance Considerations

In its current state, our work is a *usability* contribution, not a performance-improvement contribution. There are several aspects to this. `Metapool` operations impose an extra overhead on top of normal FAODEL `Pool` operations, dominated by the $O(n)$ search through all managed `Pools` for a `Pool` matching a

```
for( int i = 0; i < 25; i++ ) {
  kelpie::Key k;

  k.K1( "/metadata/" + random_string( 10 ) );

  lunasa::DataObject ldo( 0, 256, lunasa::DataObject::AllocatorType::eager );
  metapool.Publish( k, ldo );
}

for( int i = 0; i < 25; i++ ) {
  kelpie::Key k;

  k.K1( "/results/" + random_string( 10 ) );

  lunasa::DataObject ldo( 0, 256 * 1e6, lunasa::DataObject::AllocatorType::eager );
  metapool.Publish( k, ldo );
}
```

Fig. 4. An example of using the `Metapool` object to publish data to Kelpie. The application need not do anything except use the designated namespace partition for each "kind" of data it intends to publish. The `Metapool` uses the previously-supplied namespace partition functions to decide how to route the *Publish* request.

given namespace partitioning. There are obvious algorithmic and data-structure enhancements to `Metapool` which could address this, but this can be managed directly by the application through the degree of partitioning it implements. Apart from that, I/O performance will be governed by the performance of the chosen storage backend, over which FAODEL has no control.

The contribution of the `Metapool` concept, from a performance standpoint, is the degree of control it provides to applications in matching their data to the characteristics of available storage backends. We expect this to be a foundation for future performance advantages for applications using FAODEL. Instead of having to explicitly partition the data namespace according to (well-informed, to be sure) assumptions about data usage, policy-driven or machine learning approaches could be introduced to automatically micro-manage the partitioning (and consequently the IOM layer and associated storage modalities). This possibility underscores our choice of a service mediator like FAODEL as a beneficial place for such decisions, rather than performing them at the storage layer or embedding them in applications.

4 Related Work

One of the first efforts to apply semantics to hierarchical namespaces was the Intentional Naming System [2], which introduced the principle of naming what applications are interested in, as opposed to where to find them (*e.g.,* locating services by their internet hostnames). Active Names [21] was another early effort to couple resource location and naming semantics. Another example is the Proactive Directory Service [5], which allowed applications to add user-defined behavior and data management to partitions of a shared namespace. Our work

takes inspiration from these projects, giving applications tools to overlay semantics associated with how data should be persistently stored onto the shared key namespace offered by Faodel.

Many scientific simulations have the potential to generate vast quantities of output data. Domain scientists rely on sophisticated analysis and visualization to make sense of these data. Efficient use of these tools requires robust data management services to find and access output datasets. Pavlo et al. [17] compare the use of MapReduce and Parallel Database Management Systems (DBMS) for analyzing large volumes of data. For both of these approaches, data is stored and exchanged through the filesystem. SENSEI [3] defines a generic data model to facilitate the transfer of data between simulation and analysis tasks. Their generic data model is intended to simplify the process combining a simulation code with different kinds of analysis.

HPC systems have recently experienced significant growth in the depth of their storage hierarchy. SSDs, NVRAM, and 3D-stacked DRAM are all becoming increasingly common. Because each level of the hierarchy represents a different set of tradeoffs (and possibly also, different programming interfaces), application developers face an increasingly complex set of choices when decided where their data should reside. UNITY [13] provides a single interface for applications to access all levels of storage hierarchy. Data Elevator [6] provides a transparent mechanism for moving data among different layers of the storage hierarchy. Specifically, the authors describe and demonstrate their approach to moving data between burst buffers and the parallel file system.

5 Conclusion

As modern extreme computing environments evolve, flexible solutions for managing data exchanges between the applications they host will be necessary. In this paper, we have described a set of modifications to the FAODEL data management framework which allow applications to mediate among available data storage services. By partitioning the namespace provided by the Kelpie key-value service within FAODEL, applications can indicate, based on their knowledge of how data will be used, where subsets of the data they manage are best stored. We anticipate that this type of capability will prove useful in data centers where applications must make use of a set of common storage systems and services instead of being able to supply their own custom configurations. We also expect workflows which couple HPC and HPDA tasks to benefit from Kelpie's ability to persistently store data in formats suitable for off-the-shelf services without requiring explicit application data transformation or reformatting. We are working to expand the functionality of Kelpie's `Metapool` interface as well as to more fully characterize its performance with production-scale workflows.

References

1. The OpenACC application programming interface, November 2018. http://openacc-standard.org

2. Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., Lilley, J.: The design and implementation of an intentional naming system. In: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP 1999, pp. 186–201. ACM, New York (1999). https://doi.org/10.1145/319151.319164
3. Ayachit, U., et al.: The SENSEI generic in situ interface. In: Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), pp. 40–44. IEEE (2016)
4. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 66. IEEE Computer Society Press (2012)
5. Bustamante, F., Widener, P., Schwan, K.: Scalable directory services using proactivity. In: Proceedings 2002 ACM/IEEE Conference on Supercomputing. ACM/IEEE, Baltimore, November 2002
6. Dong, B., et al.: Data elevator: low-contention data movement in hierarchical storage system. In: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pp. 152–161. IEEE (2016)
7. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003. http://www.sciencedirect.com/science/article/pii/S0743731514001257. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing
8. The Apache Software Foundation: Apache cassandra (2018). https://cassandra.apache.org/. Accessed 10 May 2018
9. The Apache Software Foundation: Apache spark - unified analytics engine for big data (2018). https://spark.apache.org/. Accessed 10 May 2018
10. Germain, J.D.d.S., McCorquodale, J., Parker, S.G., Johnson, C.R.: Uintah: a massively parallel problem solving environment. In: 2000 Proceedings the Ninth International Symposium on High-Performance Distributed Computing, pp. 33–41. IEEE (2000)
11. Ghemawat, S., Dean, J.: LevelDB, a fast and lightweight key/value database library by Google (2014)
12. google: Github - google/leveldb: Leveldb is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values (2018). https://github.com/google/leveldb. Accessed 10 May 2018
13. Jones, T., et al.: Unity: unified memory and file space. In: Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2017), p. 6. ACM (2017)
14. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on C++. ACM SIGPLAN Not. **28**, 91–108 (1993)
15. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010). https://doi.org/10.1145/1773912.1773922
16. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE Computer Society (2010)
17. Pavlo, A., et al.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 165–178. ACM (2009)

18. Pébaÿ, P., et al.: Towards asynchronous many-task in situ data analysis using legion. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 1033–1037. IEEE (2016)
19. Ulmer, C., et al.: Faodel: data management for next-generation application workflows. In: Proceedings of the 9th Workshop on Scientific Cloud Computing, p. 8. ACM (2018)
20. Ulmer, C., et al.: Faodel: data management for next-generation application workflows. In: Proceedings 9th Workshop on Scientific Cloud Computing, Science Cloud 2018. ACM, June 2018
21. Vahdat, A., Dahlin, M., Anderson, T., Aggarwal, A.: Active names: flexible location and transport of wide-area resources. In: Proceedings USENIX Symposium on Internet Technology and Systems, October 1999