JCST

Vol.35 No.1 Jan. 2020

ISSN 1000-9000(Print)
/1860-4749(Online)
CODEN JCTEEM

COMPUTER

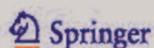
Journal of Computer Science & Technology

SPONSORED BY INSTITUTE OF COMPUTING TECHNOLOGY
THE CHINESE ACADEMY OF SCIENCES &

CHINA COMPUTER FEDERATION

SUPPORTED BY NSFC

CO-PUBLISHED BY SCIENCE PRESS &



SPRINGER

Ross R, Amvrosiadis G, Carns P *et al.* Mochi: Composing data services for high-performance computing environments. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(1): 121–144 Jan. 2020. DOI 10.1007/s11390-020-9802-0

Mochi: Composing Data Services for High-Performance Computing Environments

Robert B. Ross[1], George Amvrosiadis[2], Philip Carns[1], Charles D. Cranor[2], Matthieu Dorier[1], Kevin Harms[1]
Greg Ganger[2], Garth Gibson[3], Samuel K. Gutierrez[4], Robert Latham[1], Bob Robey[4], Dana Robinson[5]
Bradley Settlemyer[4], Galen Shipman[4], Shane Snyder[1], Jerome Soumagne[5], and Qing Zheng[2]

[1] *Argonne National Laboratory, Lemont, IL 60439, U.S.A.*

[2] *Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.*

[3] *Vector Institute for Artificial Intelligence, Toronto, Ontario, Canada*

[4] *Los Alamos National Laboratory, Los Alamos NM, U.S.A.*

[5] *The HDF Group, Champaign IL, U.S.A.*

E-mail: rross@mcs.anl.gov; gamvrosi@cmu.edu; carns@mcs.anl.gov; chuck@ece.cmu.edu; mdorier@anl.gov
harms@alcf.anl.gov; ganger@andrew.cmu.edu; garth@vectorinstitute.ai; samuel@lanl.gov
robl@mcs.anl.gov; brobey@lanl.gov; derobins@hdfgroup.org; {bws, gshipman}@lanl.gov
ssnyder@mcs.anl.gov; jsoumagne@hdfgroup.org; qingzhen@andrew.cmu.edu

Received July 1, 2019; revised November 2, 2019.

Abstract Technology enhancements and the growing breadth of application workflows running on high-performance computing (HPC) platforms drive the development of new data services that provide high performance on these new platforms, provide capable and productive interfaces and abstractions for a variety of applications, and are readily adapted when new technologies are deployed. The Mochi framework enables composition of specialized distributed data services from a collection of connectable modules and subservices. Rather than forcing all applications to use a one-size-fits-all data staging and I/O software configuration, Mochi allows each application to use a data service specialized to its needs and access patterns. This paper introduces the Mochi framework and methodology. The Mochi core components and microservices are described. Examples of the application of the Mochi methodology to the development of four specialized services are detailed. Finally, a performance evaluation of a Mochi core component, a Mochi microservice, and a composed service providing an object model is performed. The paper concludes by positioning Mochi relative to related work in the HPC space and indicating directions for future work.

Keywords storage and I/O, data-intensive computing, distributed services, high-performance computing

1 Introduction

The technologies for storing and transmitting data are in a period of rapid technological change. Nonvolatile storage technologies such as 3D NAND [1] provide new levels of performance for persistent storage, while emerging memory technologies such as 3D XPoint [2] blur the lines between memory and storage. The performance, cost, and durability of these technologies motivate the development of complex, mixed deployments to obtain the highest value. At the same time, networking technologies are also improving rapidly. The availability of high-radix routers has fostered the adoption of very low-diameter network architectures such as Dragonfly [3], Slim Fly [4], and Megafly [5] (a.k.a. Dragonfly+ [6]). Coupled with ad-

Regular Paper

Special Section on Selected I/O Technologies for High-Performance Computing and Data Analytics

This work is in part supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357; in part supported by the Exascale Computing Project under Grant No. 17-SC-20-SC, a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative; and in part supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

©Institute of Computing Technology, Chinese Academy of Sciences & Springer Nature Singapore Pte Ltd. 2020

vanced remote direct memory access (RDMA) capabilities, single-microsecond access times to remote data are feasible at the hardware level. Because of the performance sensitivity of high-performance computing (HPC) applications, HPC facilities are often early adopters of these cutting-edge technologies.

The application mix at HPC facilities and the associated data needs are undergoing change as well. Data-intensive applications have begun to consume a significant fraction of compute cycles. These applications exhibit very different patterns of data use from the checkpoint/restart behavior common with computational science simulations. Specifically, the applications rely more heavily on read access throughout the course of workflow execution, and they access and generate data more irregularly than their bulk-synchronous counterparts do.

Additionally, the *types* of data these workflows operate on differ significantly from the structured multidimensional data common in simulations: raw data from sensors and collections of text or other unstructured data are often common components. Machine learning and artificial intelligence algorithms are increasingly a component of scientific workflows as well. For example, researchers have employed the Cori system① at the National Energy Research Scientific Computing (NERSC) Center② for training models to identify new massive supersymmetric ("RPV-Susy") particles in Large Hadron Collider experiments③. As with data-intensive applications, these workloads exhibit irregular accesses, a greater prominence of read accesses, and nontraditional types of data.

Further complicating the situation is that many modern workflows are in fact a mix of these different types of activities. For example, recent work investigating optical films for photovoltaic applications employed HPC resources for a multiphase workflow [7]. The first phase of the workflow extracted candidate materials from the academic literature by using natural language processing. In the second phase, many small jobs filtered materials to remove inappropriate (e.g., poisonous) materials. The third phase relied on traditional quantum chemistry simulation, resulting in a set of materials for experimental analysis. Each phase of the workflow has its own unique data needs.

I/O Service Specialization: A Requirement of Technology and Application Diversity. The combination of rapid technological advancement and adoption along with an influx of new applications calls for the development of an ecosystem of new data services that both make efficient use of these technologies and provide high-performance implementations of the capabilities needed by applications. No single-service model (e.g., key-value (KV) store, document store, file system) is understood to meet this wide variety of needs. A report from NERSC [8] states the following:

"Ensuring that users, applications, and workflows will be ready for this transition will require immediate investment in testbeds that incorporate both new nonvolatile storage technologies and advanced object storage software systems that effectively use them. These testbeds will also provide a foundation on which a new class of data management tools can be built ...."

However, the question of how to enable the required ecosystem of such data services remains open. Without significant code reuse, the cost of each implementation is too high for an ecosystem to develop; and without significant componentization, no single implementation is likely to be readily ported from one platform to another.

Composition Model for Providing Specialized I/O Software with Mochi. Mochi directly addresses the challenges of specialization, rapid development, and ease of porting through a composition model. The Mochi framework provides a methodology and tools for communication, data storage, concurrency management, and group membership for rapidly developing distributed data services. Mochi components provide remotely accessible building blocks such as blob and KV stores that have been optimized for modern hardware. Together, these tools enable teams to quickly build new services catering to the needs of specific applications, to extract high performance from specific platforms, and to move to new platforms as they are brought online. In doing so, Mochi is enabling an ecosystem of services to develop and mature, supporting the breadth of HPC users on a wide variety of underlying technologies.

Contributions. This paper makes the following contributions: 1) presents the Mochi components and methodology for building services; 2) describes a new service composed from Mochi components; 3) evaluates a set of Mochi components and services on modern hardware.

①http://www.nersc.gov/users/computational-systems/cori/, Nov. 2019.

②https://www.nersc.gov, Nov. 2019.

③http://home.cern/topics/large-hadron-collider, Nov. 2019.

2 Mochi

In this section we describe the components of the Mochi framework, the methodology we apply for approaching the development of new services, and some of the flexibility in Mochi related to instantiating services. Table 1 lists many of the available Mochi components. At a high level, Mochi components can be separated into three groups: *core* components that provide basic functionality for developing services, *microservices* that provide a specific building block functionality, and *composed services* that are built from the core, Mochi microservices, and other libraries.

2.1 Mochi Core

At the core of the Mochi framework are three libraries that provide the tools for communicating between participants in distributed services: Mercury, Margo, and Thallium. Each of these provides different levels of functionality, and typically one of these is selected when developing a particular component. A fourth core component, Scalable Service Groups (SSG), provides a concept of a group of providers and aids in tracking membership. ABT-IO provides hooks to POSIX I/O capabilities. Lastly, Argobots [9] (developed outside the Mochi team) provides user-level thread capabilities that facilitate concurrency management.

Mercury. Mercury is a library implementing remote procedure calls (RPCs) [10]. Mercury supports the execution of remote procedures in a variety of scenarios, including over high-performance network fabrics using the libfabric[4] interface (e.g., OmniPath, InfiniBand), and abstracts the typically used notions of client and server with *origin* and *target* semantics, thereby making it particularly appropriate for use in a multiservice environment. Mercury can additionally take advantage of shared memory for execution of RPCs in the context of other processes on the same node, and facilities are included for transparently "self-executing" procedures when the target is the same process. Mixes of local and remote execution are supported, freeing the user of Mercury from having to differentiate between communication on- and off-node.

Table 1. Source Lines of Code (SLOC) for Significant Pieces of the Mochi Framework

	Component	Client	Provider	Core	Wrappers
Core	ABT-IO			784	
	Argobots			16 545	
	Mercury			29 117	
	Margo			3 894	842 (py-margo)
	Thallium			4 342	
	SSG			4 414	131 (py-ssg)
Microservices	Bake	1 355	1 800		637 (py-bake)
	POESIE	343	689		
	REMI	904	499		
	SDSKV	1 538	3 550		259 (py-sdskv)
Composed Services	DeltaFS	10 673	18 318		
	FlameStore	893	1 590		
	HEPnOS	2 929	433		
	HXHIM	7 851	6 259		
	Mobject	1 563	5 278		

Note: Implemented microservices and composed services demonstrate the small amount of code necessary to provide new capabilities within the framework.

As opposed to more standard RPC frameworks, Mercury also provides facilities to support handling of large RPC arguments (i.e., bulk data). Where possible and when natively supported by the underlying network fabric, Mercury will take advantage of RDMA and allow for data transfers to be executed from the origin's buffer to the target's memory without any additional copy (zero-copy transfers are also realized when using shared memory). Since memory patterns may vary between origin and target, Mercury also allows for *scatter-gather* types of transfers to be realized through RDMA, thereby reducing the number of required operations for

[4] https://ofiwg.github.io/libfabric/, Nov. 2019.

higher-level components. In addition to the target directly moving data via RDMA, origin remote memory descriptors can be passed on to and used by other providers in order to avoid unnecessary data transfers. This facility is leveraged in Mobject (described in Section 4).

We also note that Mercury itself does not use any threads (although libfabric internal providers may sometimes use threads to drive progress) so that higher-level components can use the threading model that is most appropriate for their use.

Margo is a C library that takes advantage of the Argobots user-level threading package to simplify the development of RPC-based services. It provides three key extensions to the Mercury RPC model. The first is a set of Argobots-aware wrappers for communication functions. These wrappers translate Mercury's event-driven asynchronous communication model into a more intuitive sequential user-level thread communication model. User-level threads that invoke these wrappers are suspended and resumed as communication operations are issued and completed. The second extension is an abstraction of the communication progress loop. This abstraction consolidates best practices for polling and communication event management into a single user-level thread that can be multiplexed on the calling process or executed on a dedicated core. Margo also provides a mechanism to spawn new user-level threads to execute handler functions for each RPC request. These threads can be redirected to different cores depending on the provider configuration to sandbox service resources. Alternatively, one can independently provision multiple providers that reside on the same daemon process.

Margo's abstraction of the communication progress loop also enables it to employ multiple polling strategies without modifying the service built atop it. The default polling strategy will use operating system mechanisms to idle gracefully until communication events occur that require processing: a strategy that conserves host CPU resources at the expense of latency. It can also optionally operate in a polling mode that continuously polls the underlying transport for activity: a strategy that consumes additional host CPU resources to improve latency. This flexibility allows Margo to be adapted to environments that prioritize resource conservation (such as colocated service deployment) or environments that prioritize performance (when dedicated service nodes are available).

Thallium. Thallium provides a C++14 library wrapping Margo that allows development of RPC-based services using all the power of modern C++. Thallium wraps all Margo, Mercury, and Argobots concepts into C++ classes with automated memory management (reference counting and cleanup) and replaces Mercury's macro-based serialization mechanism with a template-based mechanism (in a way similar to Boost's serialization library). Thallium also uses modern C++'s variadic templates to turn RPCs into callable objects that can be invoked with any kind of arguments (provided they can be serialized with the above serialization mechanism).

Scalable Service Groups. Scalable service groups provide tools for building, describing, and managing groups of providers in Mochi. The current SSG implementation leverages the SWIM [11] weakly consistent group membership protocol to detect failures and evict noncommunicating members of the group.

ABT-IO. ABT-IO provides a link between the ubiquitous POSIX file interface and the Mochi framework by allowing blocking file operations to be managed by an Argobots execution stream so that file I/O can proceed concurrent with other Mochi operations. Note that there is no corresponding core component for accessing nonvolatile memory. The persistent memory development kit (PMDK) [12] is positioned to solve this challenge, so we build on this library directly, as in the case of Bake (Subsection 2.2). Support for specific data models within the Mochi framework is implemented as microservices, rather than as part of core. This reflects an understanding that not all services will use the same data models, and so developers can incorporate only the specific microservices desired for their service. As the number and variety of supported data models grows, this design decision will become more important. Also related to data-specific service tasks, the REMI microservice (Subsection 2.2) is meant to assist in data migration both horizontally and vertically.

2.2 Mochi Microservices

The Mochi framework includes a set microservices built by using the core Mochi libraries that provide capabilities typical of distributed, data-related services. Microservices consist of client and server (or provider) libraries. The client library provides a set of functions that utilize RPC to perform work on providers.

Bake. Bake [13] began as a way to remotely store and retrieve named blobs of data using the PMDK object API with the goal of efficient storage to nonvolatile

memory backends. Since then, however, it has grown into a more general microservice for storing blobs on nonvolatile memory or file-based storage backends. We examine Bake performance in Subsection 5.2.

SDSKV. SDSKV is a recognition of the importance of KV stores in modern data management. SDSKV enables RPC-based access to multiple KV backends, including LevelDB [14], Berkeley DB [15], and in-memory databases. SDSKV is typically helpful for metadata management.

REMI. REMI (Resource Migration Interface) is a Mochi microservice designed to assist in shifting data between providers. Built using Thallium, REMI works in terms of filesets. A fileset is a group of files to be migrated from one provider to another. Given a fileset, REMI will use Mercury communication to recreate the files in the fileset on the target provider. This provides a basic capability for data management useful for constructing adaptive or hierarchical data services.

Poesie. Poesie provides the ability to embed language interpreters (e.g., Lua, Python) in Mochi services. Poesie clients can send code to Poesie providers for remote execution on their behalf, providing a flexible method of extending an RPC-based service. Currently we are evaluating Poesie as a method for enabling rapid reconfiguration of Mochi composed services.

2.3 Methodology

When developing new services using Mochi components, we have found the following methodology to be helpful in guiding design. This methodology is detailed in [16]; we summarize it here.

User Requirements. The first step is to gather requirements for users of the service, including the data model to be exposed, expected access patterns, and guarantees that the service should meet (e.g., atomicity, consistency). The data model is the data representation that the client application works with, such as NumPy arrays [17] for a machine learning application or multidimensional data for scientific simulations. Ideally our new service will work directly in terms of this data model. An important aspect of the data model is also the *namespace*, that is, the manner in which data elements are referred to and the collection of them navigated. Access pattern information identifies what types of accessor functions should be supported, as well as hinting at how data might be organized internally. For example, in a write-heavy workload, a log structured [18] organization of the data might be desirable. Guarantees made by the service further constrain the design space of the service. For example, a write-once guarantee can simplify many aspects of service implementation, whereas strict ordering of updates implies more attention being placed on how to effectively store and reconstruct the order of events.

Service Requirements. Once the user requirements have been identified, they must be translated into service requirements that describe how data will be managed by the service. These include how data and metadata will be organized and how clients will interface with the service. In terms of data organization, important considerations include what data should be distributed, whether data "objects" should be sharded (i.e., split across multiple providers), and whether data should be replicated, either for performance or for fault tolerance reasons. Similar decisions must be made for metadata, which is often treated separately from data in our model. An important aspect of metadata service requirements relates to understanding the namespace and how users will identify relevant data by using properties of metadata. Furthermore, an interface must be defined. Typically this begins with choosing a language in which the interface will be written: this is usually selected to minimize the inconvenience of access on the application side. Following this selection, a set of procedures are defined that provide the required functionality.

Implementation with Mochi Components. With requirements gathered and a protocol defined, a set of building blocks can be selected to provide the majority of functionalty, with gaps being filled with new components as needed. Mercury, Margo, or Thallium is selected to provide communication capabilities, and SSG may be employed to ease referencing of individual providers and implement provider fault detection. Other components, such as Bake and SDSKV, are typically employed to manage metadata and data. These components are composed by breaking down client procedures into constituent calls to component interfaces. When necessary, additional control components can be implemented to orchestrate more complex sequences of operations (e.g., the Mobject Sequencer (Section 4)).

2.4 Instantiating Mochi Services

By relying on the same underlying RPC mechanism, Mochi components can freely communicate between one another when on the same node or on different nodes in a system. Implementations of microservices on Margo can share the same underlying runtime, thereby execut-

ing within a single process. Further, microservice instances can be shared between composed services (e.g., a Bake instance managing node-local nonvolatile storage for more than one composed service). While we use the terms "client" and "provider" when discussing how microservice code is architected, in practice providers can act as clients and thus freely communicate with other providers. All together this flexibility means that a wide variety of client and provider placement options are available without changes to Mochi services. This includes scenarios with node local services and disaggregated ones with services on separate, network-attached resources.

3 Productively Composing Services

Mochi contributes to the rapid and productive development of specialized services in three ways. First, by supporting a variety of programming languages (i.e., C, C++, and Python), developers are able to work in their most familiar languages and with familiar supporting libraries. Second, the combination of capabilities provided by Mochi enables a wide variety of different classes of services to be developed and with a modest number of new lines of source code. Third, the Mochi design naturally separates some aspects of performance tuning from the implementation, allowing performance tuning to be approached without code modification and facilitating performance portability of services. This last point will be demonstrated in Section 5.

At the time of writing, 10 services have been developed using Mochi software components, four outside this team. In this section, we briefly describe the ones that we have been involved with. These services are described in greater detail in prior publications, and in particular the descriptions of FlameStore, HEPnOS, and ParSlice have appeared previously [16]. This material is included here to provide a more complete picture of how the Mochi methodology is applied. Each of these services provides a distinct interface for clients and leverages Mochi components in a distinct manner, speaking to the ability to develop specialized components in Mochi, and was developed in a mix of C and C++. Each has been implemented with a relatively small codebase (see Table 1) by virtue of leveraging the other components of the Mochi framework.

3.1 FlameStore

FlameStore[⑤] is a transient storage service tailored to deep learning workflows. It was developed to meet the needs of the CANDLE cancer research project[⑥]. These workflows train thousands of deep neural networks in parallel to build predictive models of drug response that can be used to optimize preclinical drug screening and drive precision-medicine-based treatments for cancer patients. Following discussions with users, FlameStore required only a few weeks of development to reach a first working version.

User Requirements. Since CANDLE workflows train deep neural networks using the Keras framework[⑦] in Python, FlameStore needs to present a Python interface capable of storing Keras models (the workflow's data model). More generally, this can be achieved by enabling storing NumPy arrays along with JSON metadata.

The workflow's access pattern consists of writing potentially large NumPy arrays. Overall, users expect such models to range from a few hundreds megabytes to a few gigabytes. These arrays are written once and never modified.

Users requested that FlameStore provide a flat namespace, that is, a simple mapping from a unique model name to a stored model. Trained models need to also be associated with a score indicating how well they perform on testing datasets. FlameStore needs to store such a score along with other user-provided metadata (including the hyperparameters used for training the model) that can be used for querying particular models. Users may also want to send Python code to nodes storing a model in order to perform local computation (e.g., evaluating some properties of the stored models in order to make decisions).

FlameStore needs to be a single-user service running for the duration of the workflow that accesses it. It needs to act like a semantic-aware distributed cache built on federated storage space (RAM, NVRAM, or disks) provided by compute nodes. It is backed up by a traditional parallel file system for persistence across multiple workflow executions.

Service Requirements. Based on the user requirements, we expect FlameStore to store few (on the order of a thousand) large objects that need to be written atomically, read atomically, and accessed locally in a

⑤https://xgitlab.cels.anl.gov/sds/flame-store, Nov. 2019.

⑥http://candle.cels.anl.gov, Nov. 2019.

⑦https://keras.io, Nov. 2019.

consistent manner. Hence we expect large data transfers to be the critical aspect of the service to optimize. We will need the storage space for these objects to be distributed. Because we need to be able to execute code within the data service to do some processing on single models, we need each model to be stored on a single node. This also aligns with the fact that workflow workers do no collectively work on the same model.

We do not expect metadata to be a bottleneck, and we can therefore use a single node to manage it. However, SDSKV is not sufficient to handle the type of queries expected from the workflow: FlameStore needs not only to store the metadata but also to make decisions on *where* to store each model, based on colocality with the node that generates it, on available space in each storage node, and on the content (semantics) of the data.

Implementation with Mochi Components. Fig.1(a) shows the organization of components used in FlameStore. Its implementation primarily relies on Bake for storage management. It uses PyMargo to implement a custom Python-based provider for semantic-aware metadata management and another custom provider for the management of storage nodes. PyBake is used to interface with Bake using Python. This Python interface also enables RDMA transfers of NumPy arrays to Bake providers. FlameStore's composition code is entirely written in Python. Fig.2(a) provides the number of lines of code used by FlameStore's components as well as the percentage this code represents: 86% of the code consists of reusable components, the remaining 14% comprising the client-side interface (6%) and the composition code and custom providers (8%). Note that this figure does not include the lines of code of Argobots (15 193) and Mercury (27 959) since these libraries existed before the Mochi project and could be replaced with alternatives in the implementation of our methodology. According to our git history, only 15 days were needed to finish a first version that users could start working with.

FlameStore enables users to plug in a *controller* module, written in Python, that implements smart data management policies. This controller makes decisions including persisting good models in HDF5 files, dis-

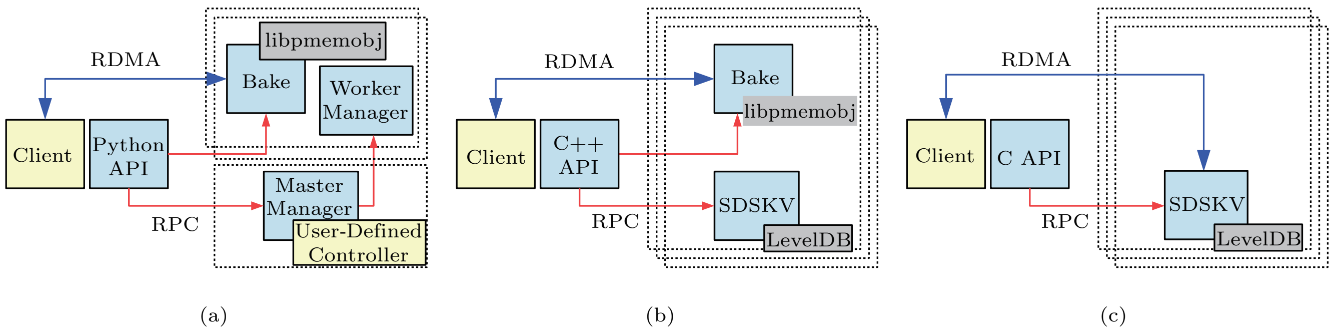


Fig.1. Architecture of our three data services [16]. The Margo runtime and some components such as MDCS and SSG have been omitted for simplicity. (a) FlameStore. (b) HEPnOS. (c) SDSDKV.

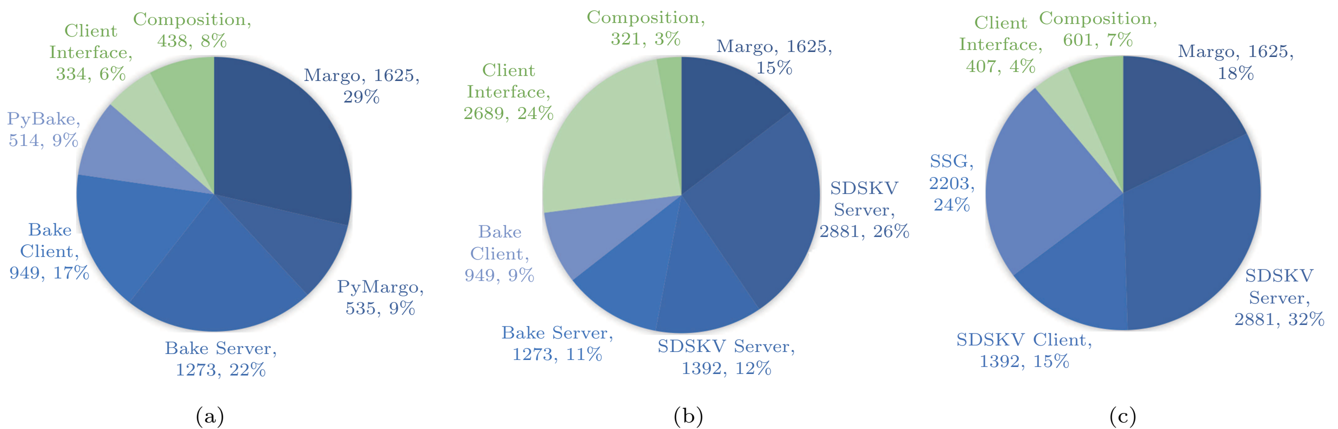


Fig.2. Single lines of code (SLOC) of the three example services, broken down into common components and custom code [16]. (a) FlameStore. (b) HEPnOS. (c) SDSDKV.

carding models that have been outperformed by other models, migrating models to improve load balancing or data locality, or compressing models that are unlikely to be reused but still need to stay in cache.

FlameStore ensures that models are written only once and atomically. It does not allow updates and partial writes. It does not replicate data by default but enables the controller to duplicate models across multiple storage locations if they need to be reused by multiple workflow workers.

Metadata in FlameStore consist of 1) a model's name, 2) a JSON-formatted model architecture, 3) the location of weight matrices (i.e., network address of the storage node and indexing information), and 4) additional user-provided metadata (e.g., the hyperparameters used for training, the accuracy of the model, and the network address of the client that is writing it). User-provided metadata are used to drive the controller's decisions.

Clients write in FlameStore by first contacting the metadata provider with the model's metadata. The metadata provider responds with the identity of a Bake provider in which to write the model. At this point the metadata provider marks the model as "pending". It is not yet visible to other clients. The client contacts the selected Bake provider, which issues RDMA pull operations to transfer the NumPy arrays from the client's memory. Upon completion, the client contacts the metadata provider again to complete the model's metadata with the location of the stored NumPy arrays.

Clients read models by contacting the metadata provider with the model's name. The metadata provider returns the model's metadata, which include the information on how to retrieve NumPy arrays from Bake providers. The metadata is sometimes the only information clients need, since it encapsulates the entire model's architecture as well as user-provided metadata. If needed, the client can request the NumPy arrays from the corresponding Bake providers, which will transfer them using RDMA push operations.

3.2 HEPnOS

HEPnOS[8] is a storage service targeting high energy physics experiments and simulations at Fermilab, and developed in the context of the SciDAC-4 "HEP on HPC" project[9].

User Requirements. Scientists at Fermilab currently use ROOT[19] files to store the massive amount of events produced by their high energy physics experiments, and also by simulations and data-processing codes. Aiming to replace ROOT to achieve better performance, better use of new technologies, and more development simplicity, we started to develop HEPnOS to specifically address their needs.

HEPnOS needs to organize data objects in a hierarchy of datasets, runs, subruns, and events. These containers act in a way similar to directories but map better to the way high-energy physics experiments organize their data. Datasets are identified by a name and can contain runs as well as other datasets. Runs, subruns, and events are identified by an integer. Runs contain subruns; subruns contain events. The notions of "relative path" and "absolute path" make it possible to address a container relative to another or relative to the root of the storage system, respectively.

Events data consist of serialized C++ data objects. Hence, HEPnOS needs to present a C++ interface that resembles that of the C++ standard library's `std::map` class, allowing to navigate items within containers using iterators. The expected access pattern is, as in FlameStore, write-once-read-many, with only atomic accesses to single objects. However, users expect a much larger number of objects (several millions). These objects, after serialization, typically range in size from a few bytes to a few kilobytes.

Service Requirements. Based on the user requirements, we defined the following service requirements. HEPnOS will need to distribute both the data and the metadata, given the large number of objects that it will store. Objects will not be sharded, but contrary to FlameStore the reason is their small size rather than that they need to be accessed locally.

Ultimately, Fermilab envisions running HEPnOS in production in a multiuser setting. In order to deal with fault-tolerance in this context, HEPnOS needs to enable both data and metadata replication. This also enables potentially better read performance.

Data and metadata will be queried based on the full path of the object; hence no particular indexing method is required.

Optimizations should also be implemented to enable bulk-loading and bulk-storing objects, in order to avoid the cumulated latency of many RPC round trips when storing or loading objects one at a time.

[8] https://xgitlab.cels.anl.gov/sds/HEPnOS, Nov. 2019.
[9] http://computing.fnal.gov/hep-on-hpc/, Nov. 2019.

Implementation with Mochi Components. Fig.1(b) shows the organization of components used in HEPnOS. HEPnOS uses Bake to store objects and SDSKV to store metadata. Typically, each service node hosts one Bake provider and one SDSKV provider, although we have not yet evaluated whether this setting is the best-performing one.

The SDSKV providers storing the information on a particular container (dataset, run, subrun, event) are selected based on the hash of the container's parent full path. Hence all the items within a given container are managed by the same set of nodes. Metadata related to serialized C++ objects, however, are managed by nodes chosen by hashing the full name of the object. This matches the expected sequential access to directory entries, versus parallel accesses to data objects.

HEPnOS also optimizes data accesses by storing small objects within their metadata, in a way similar to file systems storing data in their inodes when the data are small enough. Benchmarks should be executed on a given platform to establish the threshold below which embedding data inside metadata is advantageous.

HEPnOS bypasses Mercury's serialization mechanism and relies on Boost.Serialization instead, in order to enable serializing C++ objects with minimal changes to the user code.

Contrary to FlameStore, clients write in HEPnOS by first storing their object's data into multiple Bake providers in parallel. They then contact SDSKV providers (also selected by hashing the object's path) to store the corresponding metadata. Symmetrically, reading is done by contacting a relevant SDSKV provider, and then a relevant Bake provider.

In terms of development effort, Fig.2(b) shows that reusable components make up 63% of HEPnOS' code. The larger portion of HEPnOS's custom code is its client-side interface, which provides extensive functionalities to navigate the data store using C++ iterator patterns. The code that actually calls the Mochi components fits in a 276-line file. Our git repositories indicate that less than two months were needed between the creation of the project and the release of a first version that Fermilab could start using. While the server-side composition was ready within two weeks, most the remaining time was spent iterating on new client-side functionalities.

3.3 ParSplice

The Parallel Trajectory Splicing (ParSplice) [20] application uses a novel time-parallelization strategy for accelerated molecular dynamics. The ParSplice technique (and associated application) enables long-time scale molecular dynamics (MD) simulations of complex molecular systems by employing a Markovian chaining approach allowing many independent MD simulations to run concurrently to identify short trajectories called "segments" that are then spliced together to create a trajectory that spans long time scales. A master/worker approach is used to generate segments starting from a set of initial coordinates stored in a KV database. From these initial coordinates the workers use traditional MD simulation to generate a new segment and upon completion stores the final coordinate of the segment in a distributed KV database.

During the course of a ParSplice simulation, the KV database continues to grow to include all the states necessary for workers to generate new trajectories from a prior state. Since workers are distributed across many individual compute nodes and are stateless, the KV store must provide scalable concurrent access (read/insert). Exascale simulations using ParSplice could span tens of thousands of compute nodes with thousands of database clients accessing the KV store concurrently. To support this level of concurrency, and to minimize the memory footprint required on any one worker node, we have developed a distributed KV service, SDSDKV, built on Mochi microservices, as described in Section 2.

User Requirements. The introduction of the SDSDKV service is motivated principally by the potential reduction of code complexity in ParSplice via componentization. Moreover, this organizational strategy allows for easier runtime customization of KV service behavior (e.g., selecting an appropriate communication protocol, database back-end, or key distribution methodology [21]), thereby improving program performance portability.

The SDSDKV service needs to store values of a few thousand bytes that represent the MD state including positions, velocities, charges, and other particle characteristics. The number of KV pairs ranges from tens of thousands at current scales to several millions expected at exascale. These KV pairs are written once and never overwritten, and are accessed atomically (i.e., no partial access to a value is required). The current KV store does not erase entries, but future expansion of the service may need to remove keys.

Service Requirements. The service requirements are driven by large runs that will need to distribute the KV store across multiple nodes to balance out memory

use, access latency and bandwidth, and keep the fan-out size from a master worker within a scalable size. The objects will be distributed by their hash keys, obviating the need for metadata. Replication is an option for improving response times. This service can be asynchronous without any guarantees of determinism or handling of race conditions. The service interface needs to be implemented in C with a simple API of create/destroy for service control and put/get/delete for data handling.

Implementation with Mochi Components. Fig.1(c) shows the organization of components used in SDSDKV. SDSDKV (~1 000 SLOC) is based on the SDSKV and SSG components and on ch-placement⑩ for consistent hashing. It exposes a small, straightforward C interface providing runtime service configurability through user-supplied input parameters. SDSDKV's use centers on opaque context handles that encapsulate service-maintained state. With this design, multiple, independent SDSDKV instances may exist within a single application, each with potentially different configurations such as membership makeup, database backend type, and communication protocol used. At `sdsdkv_open()`, all members of the initializing communicator (supplied during `sdsdkv_create()`) collectively participate in service startup, initializing the individual components composing SDSDKV. From this point until context destruction `sdsdkv_put()` and `sdsdkv_get()` operations may be performed. Destinations are determined using ch-placement and serviced by the appropriate SDSKV provider.

Fig.2(c) shows the fraction of code that is reused and the fraction that is custom. Custom code includes the composition code (7%) and the client interface (4%). The client interface provides a simple, minimalistic put/get interface dispatching the operations to particular SDSKV providers based on a hash of the keys. The composition code is written in C++ and spins up multiple SDSKV providers, grouped by using SSG and distributed based on the node placement of server processes.

3.4 Other Services

DeltaFS. It is a distributed file system that runs in user space as a customizable service. Resources dedicated to DeltaFS can be customized by application workflows to provide sufficient metadata performance and in situ data indexing. The effectiveness of DeltaFS has been demonstrated across more than 130 000 CPU cores on Los Alamos National Laboratory's Trinity supercomputer, using a vector–particle-in-cell code to perform a 2-trillion particle simulation [22]. The fine-grained progress engine provided by the Mercury RPC service is critical to implementing the software-defined overlay network within DeltaFS. This overlay network is a key component of an efficient indexing pipeline that makes extremely frugal use of memory. DeltaFS also incorporates a variant of the SSG core service to track membership across ranks.

Hexadimensional Hashing Indexing Middleware (HXHIM). HXHIM provides a record-oriented I/O interface to parallel applications enabling the storage and querying of billions or trillions of small records. Building on many of the scalable approaches to metadata pioneered by DeltaFS, HXHIM provides an interface supporting the resource description framework to enable diverse workloads, including the fine-grained annotation of scientific data and scalable storage for distributed learning models. A successor to MDHIM, the multidimensional hashing indexing middleware [23], HXHIM uses the core Mochi service Thallium for RPC handling on high-speed networks, simplifying the middleware networking protocol and improving the performance and scalability of the distributed service.

Additional examples of services developed with Mochi are described in Subsection 6.1.

4 Mobject Object Storage Service

Object stores are a common alternative to cluster file systems in many environments, providing a simpler abstraction on which to layer more complex software. A number of object storage abstractions are employed at large scale today, the most popular one arguably being the S3 model⑪. For HPC, the S3 model is very restrictive: it forces writing of objects from single writing processes, and it does not allow incremental writing or overwrites, among other constraints. These restrictions limit the utility of the S3 style of object storage as a tool for managing hot data in HPC platforms. One alternative model is the RADOS [24] model, used in the Ceph [25] file system. RADOS allows for random interspersed read and write of objects, thus enabling collaborative writing of objects from multiple processes, a natural behavior in HPC applications. While the RA-

⑩https://xgitlab.cels.anl.gov/codes/ch-placement, Nov. 2019.

⑪http://aws.amazon.com/s3, Nov. 2019.

DOS model is a good fit, however, RADOS itself is not: instantiating a new instance can be difficult and time consuming, and RADOS does not make best use of HPC networks.

As a tool for exploring object storage use in HPC platforms, we have developed *Mobject*, a Mochi-based implementation of a distributed object storage service that implements a subset of the RADOS object abstraction and interface. We followed the Mochi methodology in development of the Mobject service as described below.

User Requirements. In this case, the data model to be exposed is the RADOS model, with an expectation that a variety of access patterns would be observed (e.g., as in [26]). In particular, due to the nature of HPC application access patterns, we expect to see concurrent noncontiguous writes as an important and challenging workload to be supported by the service.

Service Requirements. Due to the variety of possible deployment scales, we determined that it was necessary to allow for "scale out" by instantiating many providers. We determined that metadata would be managed within a set of KV stores and chunks of data as blobs. This includes both the namespace and metadata describing objects and their layout. To enforce RADOS access semantics a separate provider would be needed to orchestrate access, and in anticipation of noncontiguous accesses, a log-structured approach was chosen. The procedures to be implemented fall naturally out of the RADOS API.

Implementation with Mochi Components. With these decisions made, implementation could begin. Mobject (Fig.3) is implemented by using Margo, with service providers organized into a logical unit with SSG. Mobject providers consist of a new provider type, a Sequencer, that supports the RADOS operations, coupled with a Bake instance and SDSKV instance that are typically colocated on the same node and running in the context of the same process (Fig.3). The Sequencer accepts RADOS operations and maps them into corresponding Bake and SDSKV operations. The Mobject service can be scaled horizontally by instantiating any number of Mobject providers on nodes in the system. Hashing provides distribution of objects across these nodes, with all data and metadata for any given object remaining the responsibility of a single Sequencer and associated microservices.

Bake is used to store object data. Chunks of objects are stored in Bake through RDMA between Bake and client memory, and the region identifiers are returned to and tracked by the Sequencer for later reference. This approach effectively separates the control and data paths in Mobject.

SDSKV is used for three distinct tasks in Mobject.

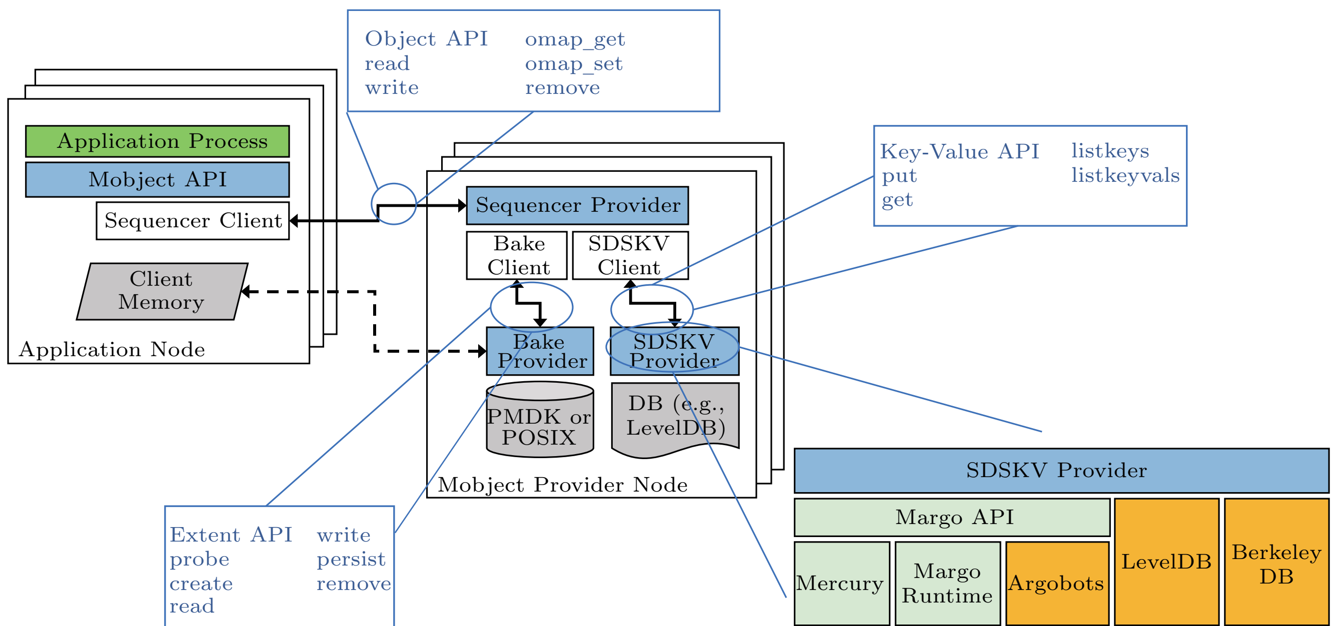


Fig.3. Mobject is a Mochi composed service providing a RADOS-like object model. The Sequencer provider accepts object operations and orchestrates their execution on the corresponding Bake and SDSKV microservices, which accept extent and key-value operations, respectively. All three services are constructed using Mochi core components: a high-level decomposition of the SDSKV provider is provided as an example.

First, SDSKV is used to track the namespace and associated object IDs, allowing the use of RADOS string object names, managing internal object IDs, and allowing bidirectional translation between these two. Second, a KV store (the *segment DB*) is used to track a log of changes made to objects. In this case the associated value is either the data itself (in the case of small changes) or a reference to a Bake region (for larger changes). The design is inspired by the Warp transactional file system [27]. Third, SDSKV also manages KV pairs representing metadata associated with objects.

Sequencer providers receive RPCs related to a specific subset of RADOS objects. For reads, the Sequencer is responsible for referencing the segment database and determining the necessary Bake operations to reconstruct the requested region, before asking Bake to transfer data into client memory. For writes, the Sequencer is responsible for selecting a timestamp for the updates, forwarding write operations to Bake to move data, and storing new records (potentially including object data) in the segment database.

5 Evaluation

In this section we evaluate a set of Mochi components and services chosen to demonstrate that the Mochi methodology and tools not only allow for rapid development but also provide opportunities for high performance to be achieved. In Subsection 5.1 we examine point-to-point latency and bandwidth in the Mochi model to demonstrate our communication capabilities. In Subsection 5.2 we consider the performance of the Bake microservice, demonstrating how Bake enables rapid storage and retrieval of blob data. In Subsection 5.3 we show how a Mochi composed service performs for a standard HPC I/O benchmark, IOR⑫. Three platforms are used in this evaluation.

Cooley⑬ is a 126-node Cray CS300 compute system at the Argonne Leadership Computing Facility⑭. Each node has 2 Intel Haswell E5-2620v3 processors with 6 cores each and 384 Gbytes of RAM. FDR InfiniBand is used as the network interconnect. The Cooley system is also attached to a number of Kove XPD external memory devices. The Kove devices provide pools of persistent memory that are connected directly to the InfiniBand fabric, offering a high-performance I/O path to applications and data services running on Cooley compute nodes. In our tests, these were configured to have an XFS file system on the device. The performance of these devices in an HPC context is explored in [28].

Bebop⑮ is a 1 024-node compute system at the Argonne Laboratory Computing Resource Center (LCRC)⑯. Bebop includes a mix of node technologies: in these tests nodes have 2 Intel Broadwell E5-2695v4 processors with 18 cores each and 128 Gbytes of RAM. Intel OmniPath is used as the network interconnect.

Cori is a mixed-architecture Cray XC40 system at NERSC. Cori includes 2 388 Intel Haswell E5-2698v3 nodes and 9 688 Intel Xeon Phi 7250 nodes. This evaluation was performed by using the Haswell nodes, each of which has 2 processors with 16 cores each and 128 Gbytes of RAM. Cray Aries is the network interconnect employed on this system.

5.1 Communication with Margo

Our first experiments demonstrate the achievable communication performance using Mochi core components across a variety of platforms, showcasing Mochi's ability to facilitate performance-portable service implementations. While MPI is not considered the best choice for service development [29], in the context of HPC systems the vendor MPI implementation can be considered as the gold standard for communication performance. For this reason we compare directly against MPI in these experiments.

The two microbenchmarks used for this study⑰ are part of the Margo regression suite. All Margo microbenchmark results are contrasted with a baseline of MPI point-to-point latency and bandwidth as measured with the OSU microbenchmarks⑱ version 5.6.1. MPI is typically the most highly optimized communication method on these platforms. Note that the OSU point-to-point bandwidth benchmark uses asynchronous MPI routines with a window size of 64 messages by default to achieve 64-way communication concurrency in the

⑫https://github.com/hpc/ior, Nov. 2019.
⑬https://www.alcf.anl.gov/user-guides/cooley, Nov. 2019.
⑭https://www.alcf.anl.gov/, Nov. 2019.
⑮https://www.lcrc.anl.gov/systems/resources/bebop. Nov. 2019.
⑯https://www.lcrc.anl.gov/, Nov. 2019.
⑰https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression, Nov. 2019.
⑱http://mvapich.cse.ohio-state.edu/benchmarks, Nov. 2019.

bandwidth tests. It uses two-sided operations, in contrast with the Margo benchmark that uses one-sided RDMA operations. Also note that the OSU latency benchmark reports one-way latency; we double this number as an estimate of round-trip latency for head-to-head comparison with the Margo benchmarks.

All benchmarks (both Margo and MPI) are executed by using a single UNIX process per node. The `numactl` utility is used to pin each process to the socket with the best network card (NIC) connectivity on each platform. The Margo benchmarks are executed with two polling modes, as described in Subsection 2.1: the default mode, which idles/sleeps when no network activity is available, and a busy-poll mode, which continuously polls the network transport. The MPI benchmarks use the MPI implementation's default busy-polling strategy in all cases.

Latency. For latency experiments, we use a microbenchmark written by using Margo that measures round-trip RPC time from the client's perspective, including encoding, checksumming, and decoding of each request and response. The provider invokes an RPC handler for each RPC. Latency statistics are reported from 100 000 sequential RPCs.

Figs.4(a)–4(c) show Margo's round-trip latency for the three platforms Cooley, Bebop, and Cori, respectively. The box-and-whisker plots show minimum, maximum, median, and first and third quartiles out of 100 000 RPCs. MPI round-trip time is plotted as a horizontal bar on these graphs using the single average latency value reported by the OSU latency benchmark. All three platforms achieve a median latency of less than 10 microseconds for Margo RPCs in busy-poll mode, despite invoking an RPC handler and performing additional encoding and checksum steps in the execution path. The median latency increases when not busy-polling, significantly so on Cori. The high increased latency on Cori is a known issue that requires additional optimization within the libfabric library. It is typical for libfabric providers to negotiate parameters such as network addressing during the first RPC, so for these tests we allow for 100 round-trip transfers before beginning to record results.

Bandwidth. For bandwidth experiments, we use a microbenchmark written by using Margo that measures sustained point-to-point throughput for RDMA payload transfers. The benchmark allocates a single large memory buffer on both the server and the client. The client then sends a single RPC instructing the server to either "pull" data from the client buffer to the server buffer or "push" data from the server buffer to the client buffer. Data is transferred by using a specified access size until a fixed amount of time has passed, and a bandwidth is calculated. We present results for a 10-second period.

The bandwidth microbenchmark also allows the caller to specify the desired level of transfer concurrency. The benchmark server spawns a corresponding number of user-level threads to issue bulk transfer operations. These user-level threads are multiplexed on a single POSIX thread. We present results for two scenarios: sequential transfers (a single user-level thread sequentially issues and completes all RDMA transfers) and 64-way concurrent transfers (64 user-level threads cooperate to issue and complete RDMA transfers).

The MPI bandwidth from the OSU benchmark is presented for comparison. The MPI bandwidth is expected to show near-theoretical peak performance because the OSU benchmark reuses a single "hot" memory buffer for all communication on both the send-

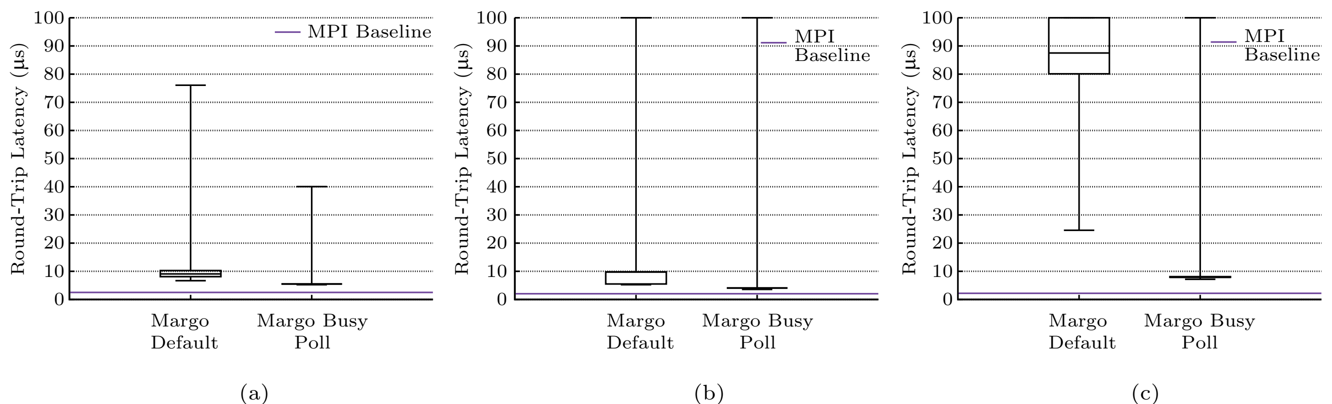
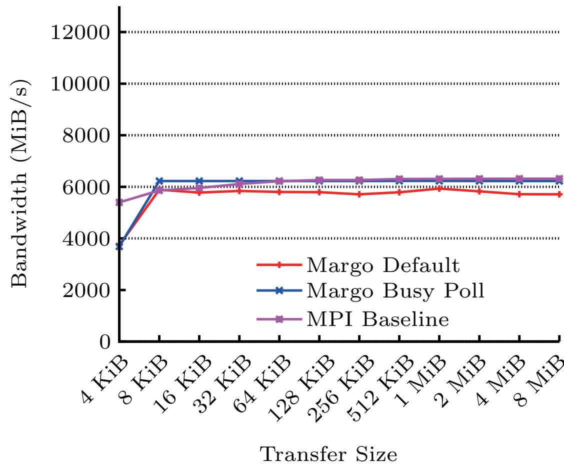
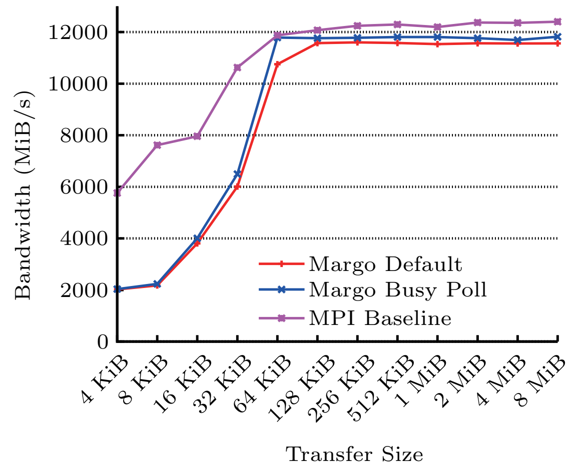


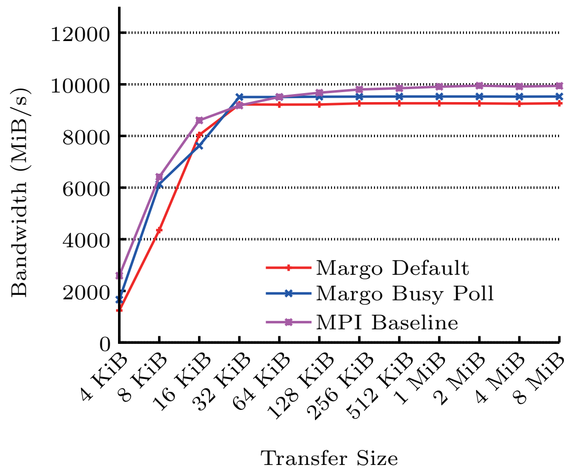
Fig.4. Margo point-to-point round-trip latency on (a) Cooley, (b) Bebop, and (c) Cori. Each system employs a different networking technology: Cooley uses FDR InfiniBand, Bebop uses OmniPath, and Cori uses Cray Aries.



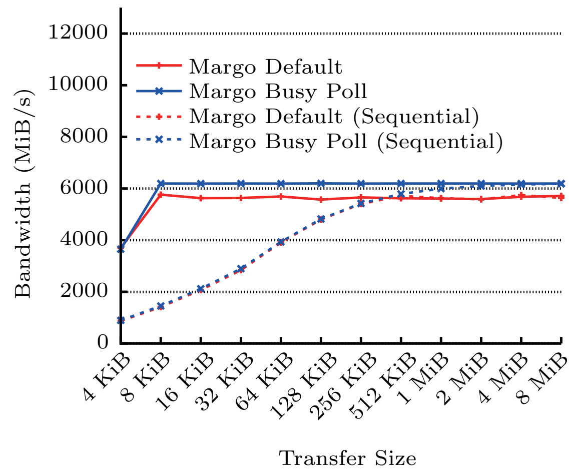
(a)



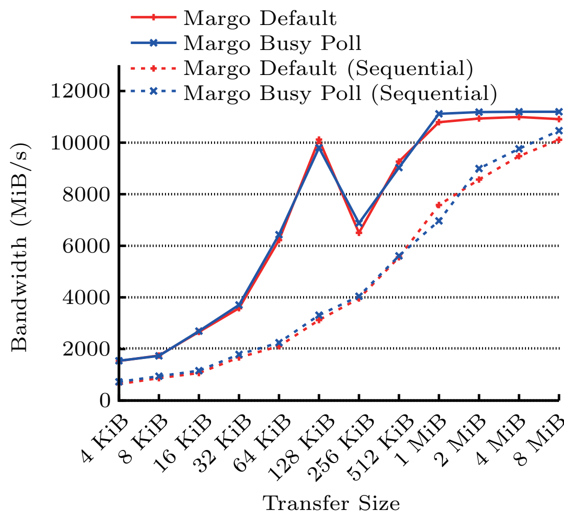
(b)



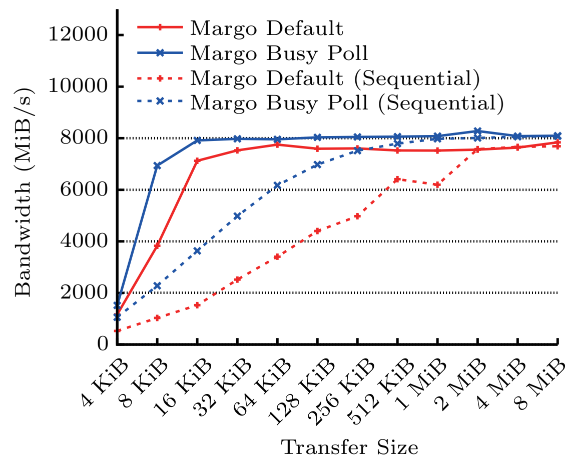
(c)



(d)



(e)



(f)

Fig.5. Margo point-to-point bandwidth in idealized and more realistic scenarios respectively on (a) (d) Cooley, (b) (e) Bebop, and (c) (f) Cori. In the idealized scenario the network is warmed up, buffers are reused, checksumming is disabled, and pages are aligned. In the realistic scenario, no warm-up iterations are performed, memory buffer reuse is minimized, checksumming enabled on control messages, and pages are not purposefully aligned. Sequential transfer speed is also shown on (d) (e) (f) using dashed lines in addition to 64-way concurrent transfer speed.

ing and receiving side. Figs.5(a)–5(c) show the Margo bandwidth with the provider pulling data on Cooley, Bebop, and Cori, respectively. In these figures we attempt to replicate the "ideal" scenario that the OSU benchmark creates by reusing a single memory buffer and allowing transfer to occur for one second before recording results to allow communication rates to stabilize.

In Figs.5(d)–5(f), the Margo benchmark deliberately iterates over "cold" memory with each transfer, to reflect a more likely data storage service workload. One-sided cold memory transfers are known to be a particular challenge on OmniPath/PSM2 (i.e., Bebop) because of its reliance on on-demand paging in the communication path. Despite these workload discrepancies, the Margo bandwidth approaches the peak MPI bandwidth on Cooley and attains at least 80% of peak performance on Bebop and Cori. While data for 64 concurrent streams is depicted for consistency with the OSU benchmark, we found that performance with only 8 concurrent streams is nearly identical. Overall these results indicate that Mochi is able to effectively exploit the high performance networks on the tested platforms, which represent three modern network technologies.

5.2 Storing Blobs with Bake

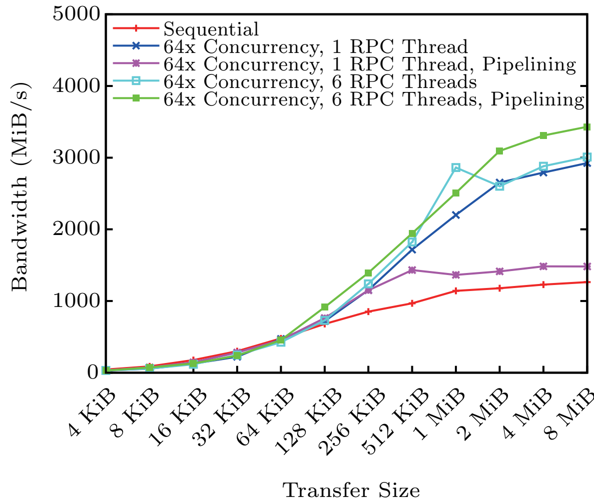
In these experiments we examine performance for the Bake microservice, focusing on the Cooley and Bebop platforms. Our goal with these experiments is to demonstrate that Mochi is able to quickly store and retrieve blobs of data on multiple platforms and backing stores.

Our test microbenchmark creates a blob on a single Bake provider and then transmits and persists data into the blob from one Bake client process, similar to the "pull" communication bandwidth results presented previously. To demonstrate both backends of Bake, we employ the PMDK backend when writing to tmpfs, and we use the POSIX backend when writing to the Kove volume on Cooley. We execute our tests over a range of client access sizes, performing 1 000 iterations of the benchmark at each access size to assist in calculating an average bandwidth value. As in the communication experiments, each benchmark is executed on a single process per node, and the `numactl` utility is used to pin each process to the optimal socket for utilizing the network card (NIC) on each platform. Busy-polling was disabled in each of these experiments.

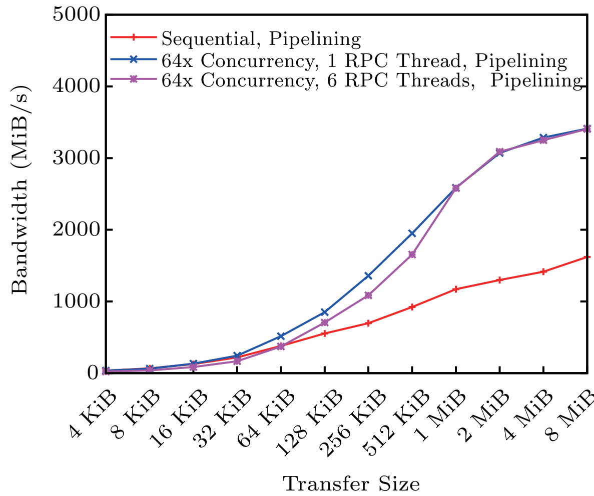
The Bake bandwidth microbenchmark also provides configurable parameters to help achieve optimal performance on different platforms or backends. In addition to options for controlling transfer concurrency and whether busy-polling is used, these benchmarks allow the user to specify the number of underlying POSIX threads to use for handling RPCs on the Bake provider and whether or not pipelining should be enabled for data transfers. Setting the number of threads to use for RPC handlers directly controls the computational resources available to the Bake provider for handling incoming client requests. When pipelining is enabled, transfers are broken down into smaller chunks and copied into preallocated buffers, with separate ULTs being spawned to drive the transfer of each chunk. In these tests, we use the default chunk size for pipelining, which is 4 MByte chunks. Note that as of this writing, pipelining is required for Bake's POSIX backend.

Figs.6(a)–6(c) present the results of these experiments. Concurrent data streams are necessary to hit peak performance, which matches with the findings from Subsection 5.1. In the best configurations, a write rate of 3 430 MiB/s is achieved on Cooley for tmpfs (59% of network peak from Margo testing), and 9 311 MiB/s is achieved on Bebop (84% of network peak from Margo testing). In the case of the tmpfs results for Cooley and Bebop, performance is additionally dependent on the number of POSIX threads allocated for handling RPCs on the Bake provider: having additional threads contributes to a 2x performance improvement on Cooley and 4x performance improvement on Bebop. These results are in contrast with the previously presented Margo communication results, which all utilized a single handler thread. Depending on the underlying storage system hardware and software, it may be increasingly important to have a number of OS threads driving progress on multiple operations in parallel in order to most effectively overlap I/O accesses and communication.

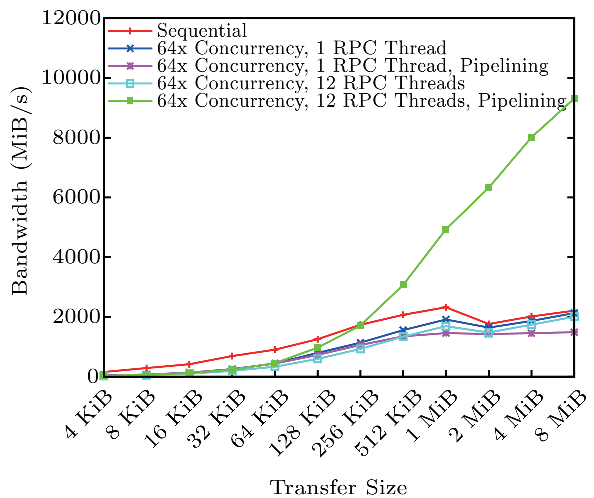
Pipelining. These results also indicate the impact Bake's pipelining mechanism has on observed I/O performance when using the PMDK backend to tmpfs. The pipelining implementation provides performance benefits in this case for two reasons. First, it further increases the concurrency of the workload by dividing it into smaller chunks that can be operated on in parallel. Second, the pipelining code path reuses intermediate data transfer buffers, avoiding potentially costly memory registrations on all client accesses, as discussed in Subsection 5.1. The Bebop results in Fig.6(c) are particularly interesting in that the best-performing configuration appears to benefit greatly from a combination



(a)



(b)



(c)

Fig.6. Bake write bandwidth on Cooley, backing to (a) RAM-backed tmpfs and (b) Kove-backed XFS volume, and (c) Bebop backing to RAM backed tmpfs. Note that scales vary on the y-axis for different platforms.

of using pipelining and additional RPC threads—enabling pipelining or using additional RPC threads has little impact on performance when done in isolation. Of special note is that the best configuration demonstrates large performance gains starting at an access size of 512 KiB, well below the pipelining unit size of 4 MiB. This further suggests that costly memory registrations are a key contributing factor to observed performance on the Omnipath/PSM2 interconnect and highlights the importance of being able to reuse preallocated buffers in the Bake microservice.

5.3 Storing Objects with Mobject

In our final experiments we examine the performance of the Mobject composed microservice on the Cooley and Cori platforms, comparing it to traditional file-based storage systems available to users on these platforms, including GPFS, Lustre, and Cray's DataWarp burst buffer system. A more complete analysis of Mobject for scientific application workloads is beyond the scope of this work and is forthcoming; the goal of these tests is to demonstrate the potential of the Mochi approach for providing high-performance services at a modest development cost. For these experiments we employ the IOR benchmark, for which we have developed a RADOS I/O backend that allows us to easily target our Mobject deployments.

We deploy Mobject over a total of 12 server nodes on Cooley and 10 server nodes on Cori in these experiments, using a single Mobject server provider per-node in each case. Bake, SDSKV, and Sequencer providers are created within the Mobject server process, with these providers sharing an RPC handler thread pool. Based on our observations in Subsection 5.2, we configured each server to use enough OS threads to efficiently overlap client communication and I/O, using 6 RPC handler threads on Cooley and 16 RPC handler threads on Cori. Pipelining was enabled in Bake in experiments where Mobject was deployed over tmpfs storage (i.e., using the PMDK-based Bake backend) and disabled in experiments where Mobject was deployed over Kove volumes (i.e., using the file-based Bake backend). The SDSKV provider was configured to use its LevelDB backend, with each KV instance stored on the same device Bake uses for storing blobs (tmpfs or a Kove volume). Busy-polling is again disabled in Margo for these experiments.

We configured IOR to use an uncoordinated workload where each process writes/reads their own unique

object (or file in the case of experiments targeting file-based storage systems). Per-process objects were sized at 1 GiB and were written in 16 MiB increments. In each experiment, we scale the number of client nodes from a single node up to the number of nodes used in our Mobject server deployment (12 nodes on Cooley, 10 nodes on Cori), with multiple clients running on each node (12 clients per-node on Cooley, 32 clients per-node on Cori). Cooley results show average, minimum, and maximum bandwidth results over 5 iterations of each test, though Cori results only show the bandwidth achieved for one iteration.

The results of these experiments are presented in Fig.7. On Cooley, the average bandwidth to tmpfs is 37 300 MiB/s for writes and 34 600 MiB/s for reads at the largest client scale, whereas the average bandwidth to the Kove volumes is 21 500 MiB/s for writes and 21 700 MiB/s for reads. This performance discrepancy between Mobject configurations is expected, as the configuration that backs to tmpfs is storing data directly in the server's RAM rather than storing in network-attached storage devices like the Kove volumes. Still, each Mobject configuration is able to easily beat the performance offered by Cooley's GPFS storage system, which caps out at an average bandwidth of 16 800 MiB/s for writes and 21 400 MiB/s for reads, and which displays an extremely high degree of performance variability relative to the Mobject results. On Cori, the average bandwidth to tmpfs is 16 600 MiB/s for writes and 38 000 MiB/s for reads at the largest client

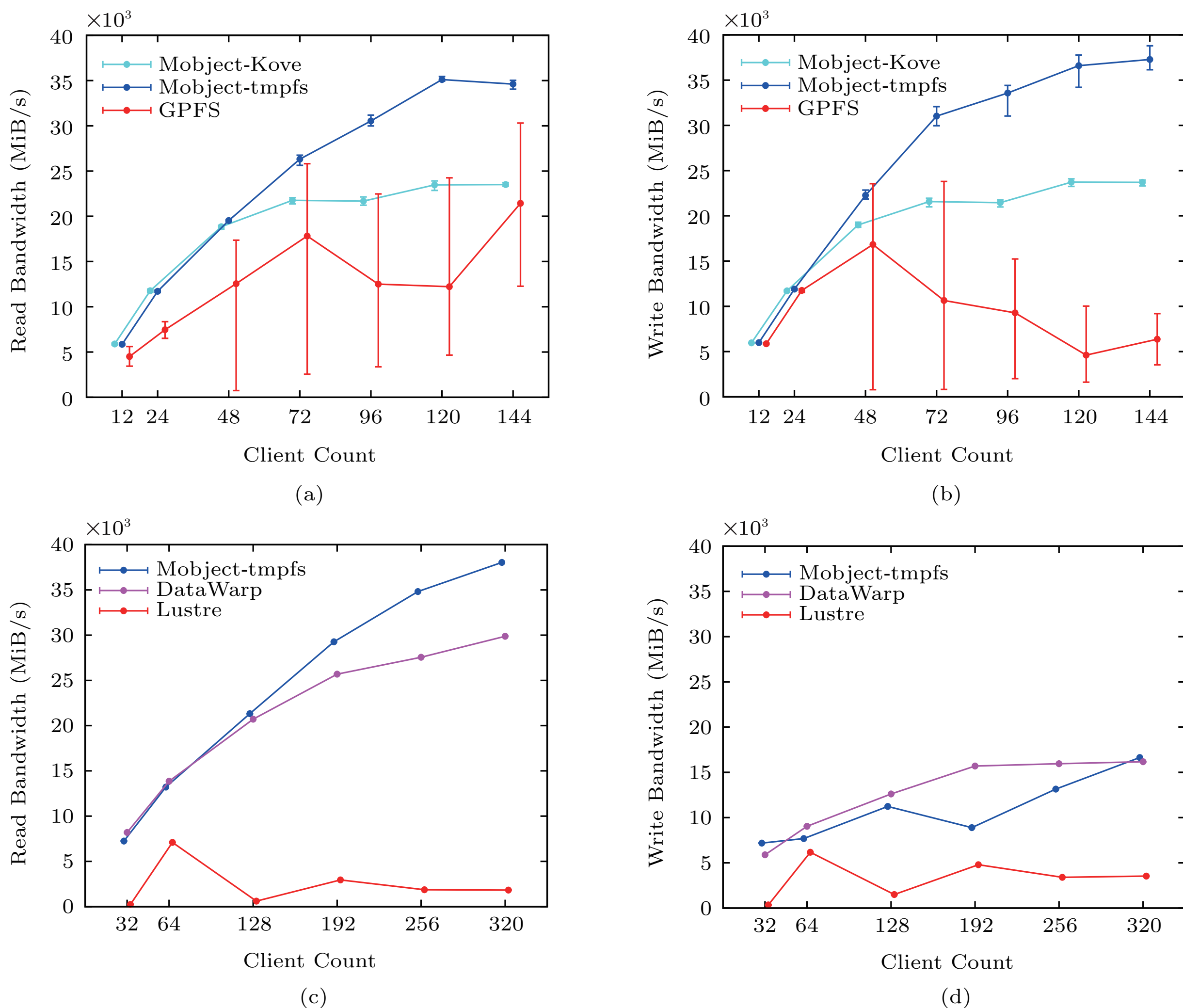


Fig.7. Mobject (a) read and (b) write bandwidth on Cooley and Mobject (c) read and (d) write bandwidth on Cori, as measured with IOR. On Cooley, results are provided for backing to both RAM-backed tmpfs and Kove-backed XFS volumes, with 12 Mobject providers used in all tests. On Cori, results are shown only for backing to RAM-backed tmpfs, with 10 providers used in all tests. Each client writes 1 GiB of data into a single object (or file). For context, GPFS file-per-process performance is shown on Cooley, while DataWarp and Lustre file-per-process performance is shown on Cori.

scale. It is not immediately clear why writes perform poorly relative to read performance, but DataWarp results exhibit the same behavior in all cases. Regardless, Mobject demonstrates a higher read performance and comparable write performance as compared with the DataWarp burst buffer storage system. Mobject results easily beat Lustre parallel file system results, much like what was observed for Cooley's GPFS results.

It is additionally clear that Mobject can offer performance comparable to the Bake results presented previously, with this performance scaling reasonably with the number of client nodes. These results further demonstrate the efficacy of the Mochi model for composing HPC data services, leveraging existing Mochi core components and microservices to compose a high-performance object storage system suitable for HPC application workloads, all with minimal development overhead.

6 Related Work

Much of the recent work most closely related to our work has been performed by teams who have adopted Mochi components. We discuss this work before touching on other related projects and technologies.

6.1 Mochi Services from the Community

Mochi components have been adopted by a number of external teams developing distributed services. These services are outlined in this section.

GekkoFS [30] implements a temporary and highly scalable file system providing relaxed POSIX semantics tailored to the majority of HPC applications. This type of specialization allows applications using the existing POSIX interface (under specific constraints) to see dramatic performance improvements as compared with file systems supporting the complete specification. The GekkoFS team has demonstrated millions of metadata operations per second, allowing it to serve applications with access patterns that were historically poor matches for file systems, and the team has shown rapid service instantiation times allowing new GekkoFS volumes to be started on a per-job basis. GekkoFS leverages Mercury and Margo in its implementation, as well as using RocksDB[19] for local KV storage.

The Unify project, the successor to BurstFS [31], similarly implements a temporary high-performance file system using local resources on nodes in the HPC system. In Unify, data are explicitly staged between the temporary Unify file system and the "permanent" parallel file system. The Unify team is exploring specialization in the form of multiple flavors of file systems, such as UnifyCR[20] for checkpoint/restart workloads and a separate specialized version for machine learning workloads. This backend specialization allows Unify to optimize for different use cases without sacrificing the portability and common toolset advantages of a POSIX interface. UnifyCR, for example, uses user-space I/O interception, scalable metadata indexing, and colocated I/O delegation to optimize bursty checkpoint workloads while still presenting a traditional file system view of the data. Unify leverages Mercury and Margo in its implementation.

Proactive Data Containers (PDC) [32] provides a data model in which a container holds a collection of objects that may reside at different levels of a potentially complex storage hierarchy and migrate between them. A PDC volume is instantiated for an application workflow and sized to meet workflow requirements for data storage and I/O. Objects can hold both streams of bytes and KV pairs, and additional metadata can be associated with objects as well. Unlike GekkoFS and UnifyCR, PDC does not present a conventional file system interface. PDC leverages Mercury for communication.

Distributed Application Object Storage (DAOS) [33] provides a transactional and multidimensional object store for use in large-scale HPC environments with embedded storage directly attached to the compute fabric. DAOS is a vendor-backed push to provide an alternative to the traditional parallel file system and has the potential to extract higher performance out of emerging low-latency storage technology. DAOS is envisioned as a multiuser and persistent volume available to all applications. It therefore encompasses a variety of system management capabilities, including distributed authentication and device provisioning. DAOS leverages Mercury in its implementation.

6.2 Other Specialized Services

HPC literature is rife with user-level service implementations. We highlight some important representatives in this subsection and note their significance.

FusionFS [34] presents a file system abstraction and

[19] https://rocksdb.org, Nov. 2019.

[20] https://github.com/LLNL/UnifyCR, Nov. 2019.

was developed to serve metadata- and write-heavy data-intensive workloads. It runs in user space and leverages a distributed hash table to hold metadata, while data are held on local storage as the data are being modified. A custom communication library is employed building on UDP. By scaling metadata operations and holding data in the system, the authors showed significant performance improvements over traditional shared parallel file systems.

DataSpaces[35] presents a virtual shared data region meant for sharing of data between applications within a workflow. A tuple model is used, and flexible key definition allows for a variety of data models to be represented. DART[36], an abstraction layer for RDMA transfers, is used for communication. Other services providing KV data models include Hercules[37], which brings memcached[38] into the HPC arena, and PayprusKV[39], which is focused on nonvolatile memory backing stores.

Spindle[40] provides an example of a specialized service related to operations. Spindle implements an overlay network used to manage parallel loading of shared libraries, removing the burden of massive bursts of read traffic from the shared file system. By implementing dynamic loader callbacks, Spindle transparently integrates into applications; and because of the nature of the data it manages, many simplifying assumptions may be made that allow for aggressive caching and data reuse.

LABIOS[41] aims to provide a single I/O service that performs well for both data-intensive and HPC applications. Building on a label model, LABIOS emphasizes support for heterogeneous resources and malleability. In the label data model, operations are transformed into tuples that represent the operation and a reference to the data. These are queued and processed by LABIOS workers independently. This model of transformation into a general "language" for I/O requests is unique and provides a method of serving different specialized frontends.

6.3 Composing Services

Three recent researches are most closely related to the concept of service composition presented here. The first, BESPOKV[42], provides a framework for defining specialized distributed KV stores. Building on a provided single-node KV store (a *datalet*), BESPOKV provides capabilities for sharding, replication, multiple consistency models, and fault tolerance. BESPOKV is evaluated by using a variety of workloads that demonstrate the value of different back end KV stores and the flexibility of BESPOKV. BESPOKV highlights the value of componentization and of providing group membership and management functionality as part of a service development framework, but it is limited to the KV data model.

The second, Faodel[43], provides a set of services for data management and exchange in HPC workflows. Three major components of Faodel are Kelpie, Opbox, and Lunasa. Kelpie provides a key-blob abstraction, essentially a KV abstraction where the service is unaware of the contents or semantics of the value. Keys can be selected by users of the service for implicit data sharing or can be shared explicitly. Opbox implements communication by representing protocols as state machines, aiming to support more complex communication patterns than simple pairwise communication. A name service implementation is included as part of Opbox. Lunasa is a memory management service to aid in using RDMA transfers. Similar to BESPOKV, Faodel focuses on a KV abstraction for data; but distinct from BESPOKV, it does not emphasize having multiple backends catering to different use cases. The Opbox component distinguishes Faodel and provides an ability to construct services such as monitoring systems that have fan-in patterns of communication.

Malacology[44] examines composition by *decomposing* an existing service, Ceph, to make it programmable. The authors recognized the inherent value of production quality software and seek to leverage this in new contexts by implementing public interfaces to what originally were internal interfaces. They discussed challenges related to this approach and demonstrated how components from Ceph can be used to build alternative services. The underlying subsystems they identify and make available (object access, cluster monitoring, and metadata management and balancing) have some analogies in Mochi, but in particular the metadata service maps more directly to file-oriented services.

7 Conclusions and Future Work

Rapid technological developments coupled with the diversification of workflows on HPC systems mandate the development of new, specialized data services that make the most effective use of new systems and are highly productive for users. In this paper we have described the Mochi framework, components, and methodology for developing distributed services

and demonstrated their use on modern HPC platforms. Mochi directly addresses the challenges of specialization, rapid development, and ease of porting through a composition model, providing a methodology and tools for communication, managing concurrency, and group membership for rapidly developing distributed services with a relatively small amount of new code. The Mochi framework is open source, actively being developed and extended, and available online[21].

We anticipate the need for rapid online adaptation of distributed services in response to application needs and environmental change. With this in mind, we are investigating what building blocks would be most effective for implementing distributed control for services built in the Mochi framework.

Acknowledgments

We thank Brian Toonen for his help in configuring the Kove devices on Cooley for this work.

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract No. DE-AC02-06CH11357. This research also used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. Moreover, this research used computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] Venkatesan S, Aoulaiche M. Overview of 3D NAND technologies and outlook invited paper. In *Proc. the 2018 Non-Volatile Memory Technology Symposium*, Oct. 2018, Article No. 15.

[2] Hady F T, Foong A, Veal B, Williams D. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE*, 2017, 105(9): 1822-1833.

[3] Kim J, Dally W J, Scott S, Abts D. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Comput. Architecture News*, 2008, 36(3): 77-88.

[4] Besta M, Hoeer T. Slim Fly: A cost effective low-diameter network topology. In *Proc. the Int. Conf. for High Performance Comput., Networking, Storage and Anal.*, November 2014, pp.348-359.

[5] Flajslik M, Borch E, Parker M A. Megafly: A topology for exascale systems. In *Proc. the 33rd International Conference on High Performance Computing*, June 2018, pp.289-310.

[6] Shpiner A, Haramaty Z, Eliad S, Zdornov V, Gafni B, Zahavi E. Dragonfly+: Low cost topology for scaling datacenters. In *Proc. the 3rd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era*, February 2017, pp.1-8.

[7] Sivaraman G, Beard E, Vazquez-Mayagoitia A, Vishwanath V, Cole J. UV/vis absorption spectra database auto-generated for optical applications via the Argonne data science program. In *Proc. the 2019 APS March Meeting*, March 2019.

[8] Lockwood G K, Hazen D, Koziol Q *et al.* Storage 2020: A vision for the future of HPC storage. Technical Report, National Energy Research Scientific Computing Center, 2017. https://escholarship.org/content/qt744479dp/qt744479d-p.pdf, Sept. 2019.

[9] Seo S, Amer A, Balaji P *et al.* Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 2018, 29(3): 512-526.

[10] Soumagne J, Kimpe D, Zounmevo J, Chaarawi M, Koziol Q, Afsahi A, Ross R. Mercury: Enabling remote procedure call for high-performance computing. In *Proc. the 2013 IEEE International Conference on Cluster Computing*, September 2013, Article No. 50.

[11] Das A, Gupta I, Motivala A. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Proc. the 2002 International Conference on Dependable Systems and Networks*, June 2002, pp.303-312.

[12] Rudoff A. Persistent memory programming. *Login: The Usenix Magazine*, 2017, 42(2): 34-40.

[13] Carns P, Jenkins J, Cranor C, Atchley S, Seo S, Snyder S, Hoeer T, Ross R. Enabling NVM for data-intensive scientific services. In *Proc. the 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, November 2016, Article No. 4.

[14] Ghemawat S, Dean J. LevelDB — A fast and lightweight key/value database library by Google. https://github.com/google/leveldb, Sept. 2019.

[15] Olson M A, Bostic K, Seltzer M I. Berkeley DB. In *Proc. the 1999 USENIX Annual Technical Conference*, June 1999, pp.183-191.

[16] Dorier M, Carns P, Harms K *et al.* Methodology for the rapid development of scalable HPC data services. In *Proc. the 3rd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems*, November 2018, pp.76-87.

[17] van der Walt S, Colbert S C, Varoquaux G. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 2011, 13(2): 22-30.

[18] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 1992, 10(1): 26-52.

[19] Brun R, Rademakers F. ROOT — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 1997, 389(1/2): 81-86.

[21] https://www.mcs.anl.gov/research/projects/mochi/, Nov. 2019.

[20] Perez D, Cubuk E D, Waterland A, Kaxiras E, Voter A F. Long-time dynamics through parallel trajectory splicing. *Journal of Chemical Theory and Computation*, 2015, 12(1): 18-28.

[21] Sevilla M A, Maltzahn C, Alvaro P, Nasirigerdeh R, Settlemyer B W, Perez D, Rich D, Shipman G M. Programmable caches with a data management language and policy engine. In *Proc. the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2018, pp.203-212.

[22] Zheng Q, Cranor C D, Guo D H, Ganger G R, Amvrosiadis G, Gibson G A, Settlemyer B W, Grider G, Guo F. Scaling embedded in-situ indexing with deltaFS. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2018, Article No. 3.

[23] Greenberg H, Bent J, Grider G. MDHIM: A parallel key/value framework for HPC. In *Proc. the 7th USENIX Workshop on Hot Topics in Storage and File Systems*, July 2015, Article No. 10.

[24] Weil S A, Leung A W, Brandt S A, Maltzahn C. RADOS: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proc. the 2nd International Petascale Data Storage Workshop*, November 2007, pp.35-44.

[25] Weil S A, Brandt S A, Miller E L, Long D D E, Maltzahn C. Ceph: A scalable, high-performance distributed file system. In *Proc. the 7th USENIX Symposium on Operating Systems Design and Implementation*, November 2006, pp.307-320.

[26] Liu J L, Koziol Q, Butler G F, Fortner N, Chaarawi M, Tang H J, Byna S, Lockwood G K, Cheema R, Kallback-Rose K A, Hazen D, Prabhat. Evaluation of HPC application I/O on object storage systems. In *Proc. the 3rd IEEE/ACM International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems*, November 2018, pp.24-34.

[27] Escriva R, Sirer E G. The design and implementation of the warp transactional file system. In *Proc. the 13th USENIX Symposium on Networked Systems Design and Implementation*, March 2016, pp.469-483.

[28] Kunkel J, Betke E. An MPI-IO in-memory driver for non-volatile pooled memory of the Kove XPD. In *Proc. the 2017 International Workshops on High Performance Computing*, June 2017, pp.679-690.

[29] Latham R, Ross R B, Thakur R. Can MPI be used for persistent parallel services? In *Proc. the 13th European PVM/MPI Users' Group Meeting*, September 2006, pp.275-284.

[30] Vef M A, Moti N, Süß T, Tocci T, Nou R, Miranda A, Cortes T, Brinkmann A. GekkoFS — A temporary distributed file system for HPC applications. In *Proc. the 2018 IEEE International Conference on Cluster Computing*, September 2018, pp.319-324.

[31] Wang T, Mohror K, Moody A, Sato K, Yu W K. An ephemeral burst-buffer file system for scientific applications. In *Proc. the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2016, pp.807-818.

[32] Tang H J, Byna S, Tessier F *et al.* Toward scalable and asynchronous object-centric data management for HPC. In *Proc. the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2018, pp.113-122.

[33] Intel Corporation. DAOS: Revolutionizing high-performance storage with Intel Optane technology. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/high-performance-storage-brief.pdf, June 2019.

[34] Zhao D F, Zhang Z, Zhou X B, Li T L, Wang K, Kimpe D, Carns P, Ross R, Raicu I. FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Proc. the 2014 IEEE International Conference on Big Data*, October 2014, pp.61-70.

[35] Docan C, Parashar M, Klasky S. DataSpaces: An interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 2011, 15(2): 163-181.

[36] Docan C, Parashar M, Klasky S. Enabling high-speed asynchronous data extraction and transfer using DART. *Concurrency and Computation: Practice and Experience*, 2010, 22(9): 1181-1204.

[37] Duro F R, Blas J G, Isaila F, Pérez J C, Wozniak J M, Ross R. Exploiting data locality in Swift/T workflows using Hercules. In *Proc. the 1st Network for Sustainable Ultrascale Computing Workshop*, October 2014.

[38] Fitzpatrick B. Distributed caching with Memcached. *Linux Journal*, 2004, 2004(124): 72-76.

[39] Kim J, Lee S, Vetter J S. PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures. In *Proc. the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017, Article No. 57.

[40] Frings W, Ahn D H, LeGendre M, Gamblin T, de Supinski B R, Wolf F. Massively parallel loading. In *Proc. the 27th International ACM Conference on International Conference on Supercomputing*, June 2013, pp.389-398.

[41] Kougkas A, Devarajan H, Lofstead J, Sun X H. LABIOS: A distributed label-based I/O system. In *Proc. the 28th International Symposium on High-Performance Parallel and Distributed Computing*, June 2019, pp.13-24.

[42] Anwar A, Cheng Y, Huang H, Han J, Sim H, Lee D, Douglis F, Butt A R. BESPOKV: Application tailored scale-out key-value stores. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2018, Article No. 2.

[43] Ulmer C, Mukherjee S, Templet G, Levy S, Lofstead J, Widener P, Kordenbrock T, Lawson M. Faodel: Data management for next-generation application workflows. In *Proc. the 9th Workshop on Scientific Cloud Computing*, June 2018, Article No. 8.

[44] Sevilla M A, Watkins N, Jimenez I, Alvaro P, Finkelstein S, LeFevre J, Maltzahn C. Malacology: A programmable storage system. In *Proc. the 12th European Conference on Computer Systems*, April 2017, pp.175-190.

Robert B. Ross is a senior computer scientist at Argonne National Laboratory, Lemont, and a senior fellow at the Northwestern–Argonne Institute for Science and Engineering at Northwestern University, Evanston. Dr. Ross's research interests are in system software and architectures for high-performance computing and data analysis systems, in particular storage systems and software for I/O and message passing. Rob received his Ph.D. degree in computer engineering from Clemson University in 2000. Rob was a recipient of the 2004 Presidential Early Career Award for Scientists and Engineers.

George Amvrosiadis is an assistant research professor of electrical and computer engineering and, by courtesy, computer science at Carnegie Mellon University, Pittsburgh, and a member of the Parallel Data Laboratory. His current research focuses on distributed and cloud storage, new storage technologies, high-performance computing, and storage for machine learning. He co-teaches courses on cloud computing and storage systems.

Philip Carns is a principal software development specialist in the Mathematics and Computer Science Division of Argonne National Laboratory, Lemont. He is also an adjunct associate professor of electrical and computer engineering at Clemson University and a fellow of the Northwestern-Argonne Institute for Science and Engineering. His research interests include characterization, modeling, and development of storage systems for data-intensive scientific computing.

Charles D. Cranor is a senior systems scientist in the Parallel Data Laboratory at Carnegie Mellon University, Pittsburgh. Chuck's research interests are currently focused on operating systems, storage systems, and high-performance computing.

Matthieu Dorier is a software development specialist in the Mathematics and Computer Science Division of Argonne National Laboratory, Lemont. He obtained his Ph.D. degree from Ecole Normale Superieure de Rennes, France. His research interests include distributed and parallel storage systems for high-performance and data-intensive scientific computing, in situ analysis and visualization for HPC simulations, and collective communication algorithms.

Kevin Harms is a performance engineering team lead in the Argonne Leadership Computing Facility of Argonne National Laboratory, Lemont, who specializes in high-performance storage and I/O.

Greg Ganger is the Jatras Professor of electrical and computer engineering at Carnegie Mellon University (CMU) and Director of the Parallel Data Laboratory (www.pdl.cmu.edu), which is CMU's storage systems and large-scale infrastructure research center. His Ph.D. degree in computer science and engineering is from the University of Michigan. He has broad research interests in computer systems, with current projects exploring system support for large-scale ML (Big Learning), resource management in cloud computing, and software systems for heterogeneous storage clusters, HPC storage, and NVM.

Garth Gibson is a co-author of a best paper at SC14 on scalable metadata for PFSs, was instrumental in the Development of the Parallel Log Structured File System, co-founded USENIX's Conference on File and Storage Technologies and SIGHPC's Workshop on Parallel Data Storage, founded and architected Panasas' PanFS products, founded Carnegie Mellon's Parallel Data Laboratory, and in 1988 co-authored a pioneering paper on RAID.

Samuel K. Gutierrez is a computer scientist in the High Performance Computing Division at Los Alamos National Laboratory, Los Alamos. He attended New Mexico Highlands University in Las Vegas, NM, where he received his Bachelor of Science degree in computer science. He received his Master's degree and Ph.D. degree in computer science from the University of New Mexico in 2009 and 2018, respectively. Gutierrez's research is focused primarily on the design and construction of low-level software systems that allow parallel and distributed programs to run more efficiently on supercomputers.

Robert Latham, as a principle software development specialist at Argonne National Laboratory, Lemont, strives to make scientific applications use I/O more efficiently. After earning his B.S. (1999) and M.S. (2000) degrees in computer engineering at Lehigh University (Bethlehem, PA), he worked at Paralogic, Inc., a Linux cluster start-up. His work with cluster software including MPI implementations and parallel file systems eventually led him to Argonne, where he has been since 2002. His research focus has been on high-performance IO for scientific applications and IO metrics. He has worked on the ROMIO MPI-IO implementation, the parallel file systems PVFS (v1 and v2), Parallel NetCDF, and Mochi I/O services.

Bob Robey is a research scientist in the Eulerian Applications group at Los Alamos National Laboratory, Los Alamos. He has over 20 years of experience in shock wave research including the operation of large explosively driven shock tubes and writing compressible fluid dynamics codes. He helped establish the High Performance Computing Center at the University of New Mexico. He co-led the 2011 X Division Summer Workshop focus group on Exascale applications.

Dana Robinson is a senior software engineer and scientific data consultant at The HDF Group, Champaign. His interests are scientific data storage and modeling, particularly in bioinformatics.

Bradley Settlemyer is a senior scientist in Los Alamos National Laboratory's HPC Design group. He received his Ph.D. degree in computer engineering from Clemson University in 2009 with a research focus on the design of parallel file systems. He currently leads the storage systems research efforts within Los Alamos' Ultrascale Research Center and his team is responsible for designing and deploying state-of-the-art storage systems for enabling scientific discovery. He is the Primary Investigator on projects ranging from ephemeral file system design to archival storage systems using molecular information technology and he has published papers on emerging storage systems, long distance data movement, system modeling, and storage system algorithms.

Galen Shipman is a senior scientist at Los Alamos National Laboratory (LANL) focusing on research and development of advanced technologies for large-scale HPC applications. He currently leads a team within the Applied Computer Science group at LANL and is Principal Investigator (PI)/co-PI on project working on runtime systems, storage systems and I/O, and programming models. Working at the intersection of computer science and computational science, he has a passion for interdisciplinary research and development in HPC.

Shane Snyder has an M.S. degree in computer engineering from Clemson University. He has been involved in the CODES (Enabling Co-Design of Multilayer Exascale Storage) and TOKIO (Total Knowledge of I/O) projects. He also is on the Darshan team received a 2018 R&D 100 award.

Jerome Soumagne received his Master's degree in computer science in 2008 from ENSEIRB Bordeaux and his Ph.D. degree in computer science in 2012 from the University of Bordeaux in France while being an HPC software engineer at the Swiss National Supercomputing Centre in Switzerland. He is now a lead HPC software engineer at the HDF Group. His research interests include large scale I/O problems, parallel coupling and data transfer between HPC applications and in-transit/in-situ analysis and visualization. He is responsible for the development of the Mercury RPC interface and an active developer of the HDF5 library.

Qing Zheng is a Ph.D. student advised by Garth Gibson and George Amvrosiadis in the Computer Science Department of Carnegie Mellon University. His research focuses on distributed filesystem metadata and storage systems.

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

Volume 35, Number 1, January 2020

Special Section on Selected I/O Technologies for High-Performance Computing and Data Analytics

Preface .... *Xian-He Sun and Weikuan Yu* (1)

Ad Hoc File Systems for High-Performance Computing .... *André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip Carns, Toni Cortes, Scott A. Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, and Marc-André Vef* (4)

Design and Implementation of the Tianhe-2 Data Storage and Management System .... *Yu-Tong Lu, Peng Cheng, and Zhi-Guang Chen* (27)

Lessons Learned from Optimizing the Sunway Storage System for Higher Application I/O Performance .... *Qi Chen, Kang Chen, Zuo-Ning Chen, Wei Xue, Xu Ji, and Bin Yang* (47)

Gfarm/BB — Gfarm File System for Node-Local Burst Buffer .... *Osamu Tatebe, Shukuko Moriwake, and Yoshihiro Oyama* (61)

GekkoFS — A Temporary Burst Buffer File System for HPC Applications .... *Marc-André Vef, Nafiseh Moti, Tim Süß, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann* (72)

I/O Acceleration via Multi-Tiered Data Buffering and Prefetching .... *Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun* (92)

Mochi: Composing Data Services for High-Performance Computing Environments .... *Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng* (121)

ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems .... *Suren Byna, M. Scot Breitenfeld, Bin Dong, Quincey Koziol, Elena Pourmal, Dana Robinson, Jerome Soumagne, Houjun Tang, Venkatram Vishwanath, and Richard Warren* (145)

Special Section on Applications

SmartPipe: Towards Interoperability of Industrial Applications via Computational Reflection .... *Su Zhang, Hua-Qian Cai, Yun Ma, Tian-Yue Fan, Ying Zhang, and Gang Huang* (161)

Labeled Network Stack: A High-Concurrency and Low-Tail Latency Cloud Server Framework for Massive IoT Devices .... *Wen-Li Zhang, Ke Liu, Yi-Fan Shen, Ya-Zhu Lan, Hui Song, Ming-Yu Chen, and Yuan-Fei Chen* (179)

CirroData: Yet Another SQL-on-Hadoop Data Analytics Engine with High Performance .... *Zheng-Hao Jin, Haiyang Shi, Ying-Xin Hu, Li Zha, and Xiaoyi Lu* (194)

A Case for Adaptive Resource Management in Alibaba Datacenter Using Neural Networks .... *Sa Wang, Yan-Hai Zhu, Shan-Pei Chen, Tian-Ze Wu, Wen-Jie Li, Xu-Sheng Zhan, Hai-Yang Ding, Wei-Song Shi, and Yun-Gang Bao* (209)

AquaSee: Predict Load and Cooling System Faults of Supercomputers Using Chilled Water Data .... *Yu-Qi Li, Li-Quan Xiao, Jing-Hua Feng, Bin Xu, and Jian Zhang* (221)

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

《计算机科学技术学报》

Volume 35 Number 1 2020 (Bimonthly, Started in 1986)

Indexed in: **SCIE, Ei, INSPEC, JST, AJ, MR, CA, DBLP**

Edited by:

THE EDITORIAL BOARD OF JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

Guo-Jie Li, Editor-in-Chief, P.O. Box 2704, Beijing 100190, P.R. China

Managing Editor: Feng-Di Shu E-mail: jcst@ict.ac.cn http://jcst.ict.ac.cn Tel.: 86-10-62610746

Copyright ©Institute of Computing Technology, Chinese Academy of Sciences and Springer Nature Singapore Pte Ltd. 2020

Sponsored by: Institute of Computing Technology, CAS & China Computer Federation

Supervised by: Chinese Academy of Sciences

Undertaken by: Institute of Computing Technology, CAS

Published by: Science Press, Beijing, China

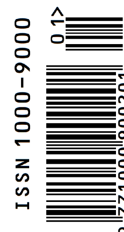
Printed by: Beijing Kexin Printing House

Distributed by:

China: All Local Post Offices

Other Countries: Springer Nature Customer Service Center GmbH, Tiergartenstr. 15, 69121 Heidelberg, Germany

Available Online: https://link.springer.com/journal/11390



国内统一刊号: CN11-2296/TP 国内邮发代号: 2-578 RMB ¥150.00