SAND2019-9926C

LABIOS: A Distributed Label-Based I/O System

Anthony Kougkas, Hariharan Devarajan, Jay Lofstead*, and Xian-He Sun
Illinois Institute of Technology - Department of Computer Science, *Sandia National Laboratories
akougkas@hawk.iit.edu,hdevarajan@hawk.iit.edu,gflofst@sandia.gov,sun@iit.edu

ABSTRACT

In the era of data-intensive computing, large-scale applications, in both scientific and the BigData communities, demonstrate unique I/O requirements leading to a proliferation of different storage devices and software stacks, many of which have conflicting requirements. In this paper, we investigate how to support a wide variety of conflicting I/O workloads under a single storage system. We introduce the idea of a *Label*, a new data representation, and, we present LABIOS: a new, distributed, Label- based I/O system. LABIOS boosts I/O performance by up to 17x via asynchronous I/O, supports heterogeneous storage resources, offers storage elasticity, and promotes in-situ analytics via data provisioning. LABIOS demonstrates the effectiveness of storage bridging to support the convergence of HPC and BigData workloads on a single platform.

CCS CONCEPTS

•Information systems → Distributed storage; Hierarchical storage management; *Storage power management;* **•Computer systems organization →** *Distributed architectures; Data flow architectures;* **Heterogeneous (hybrid) systems;**

KEYWORDS

Label-based I/O, Storage Bridging, Heterogeneous I/O, Datalabels, Task-based I/O, Exascale I/O, Energy-aware I/O, Elastic Storage

ACM Reference format:
Anthony Kougkas, Hariharan Devarajan, Jay Lofstead*, and Xian-He Sun . 2019. LABIOS: A Distributed Label-Based I/O System. In *Proceedings of The 28th International Symposium on High-Performance Parallel and Distributed Computing, Phoenix, AZ, USA, June 22–29, 2019 (HPDC '19),* 12 pages.
DOI: 10.1145/3307681.3325405

1 INTRODUCTION

Large-scale applications, in both scientific and the BigData communities, demonstrate unique I/O requirements that none of the existing storage solutions can unequivocally address them. This has caused a proliferation of different storage devices, device placements, and software stacks, many of which have conflicting requirements. Each new architecture has been accompanied by new software for extracting performance on the target hardware. Further, to reduce the I/O performance gap, hardware composition of modern storage systems is going through extensive changes by adding new storage devices. This leads to heterogeneous storage resources where data movement is complex, expensive, and dominating the performance of most applications [36]. For instance, machines with a large amount of RAM allow new computation frameworks, such as Apache Spark [72], to thrive. Supercomputers equipped with node-local fast storage, such as NVMe drives, take scientific simulation to new performance standards [6]. To achieve computational efficiency modern parallel and distributed storage systems must efficiently support a diverse and conflicting set of features.

Data-intensive applications grow more complex as the volume of data increases, creating diverse I/O workloads. Thus, the features a distributed storage system is required to support also increases dramatically in number and are often conflicting. For instance, scientific applications demonstrate a periodic behavior where computations are followed by intense I/O phases. Highly-concurrent write-intensive workloads (e.g., final results, checkpoints), shared file parallel access, frequent in-place data mutations, and complex data structures and formats are the norm in most High-Performance Computing (HPC) workloads [35]. On the other hand, iterative write-once, read-many data access, created by the popular MapReduce paradigm, are the defacto patterns in most BigData applications [50]. Another example is the ability of an I/O subsystem to handle data mutations. In HPC, the ability to frequently update data forces storage systems to obey certain standards, such as POSIX, and increase the cost of metadata operations which is projected to limit the scalability of these systems [57]. In contrast, most cloud storage solutions prefer an immutable representation of data, such as RDDs [71] or key-value pairs. Finally, each application manipulates data in a different data representation (i.e., format) spanning from files, objects, buckets, key-value pairs, etc., which increases the complexity of the data organization inside a storage system. To navigate this vast and diverse set of contradictory I/O requirements, the software landscape is filled with custom, highly specialized storage solutions varying from high-level I/O libraries to custom data formats, interfaces, and, ultimately, storage systems.

The ability to seamlessly execute different conflicting workloads is a highly desirable feature. However, the tools and cultures of HPC and BigData have diverged, to the detriment of both [56], and unification is essential to address a spectrum of major research domains. This divergence has led organizations to employ separate computing and data analysis clusters. For example, NASA's Goddard Space Flight Center uses one cluster to conduct climate simulation, and another one for the data analysis of the observation data [76]. Due to the data copying between the two clusters, the data analysis is currently conducted off-line, not at runtime. The data transfer between storage systems along with any necessary data transformations are a serious performance bottleneck and cripples the productivity of those systems [37]. Additionally, it increases the wastage of energy and the complexity of the workflow. Integrating analytics into a large scale simulation code has been proven to significantly boost performance and can lead to more accurate and

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
DOI: http://dx.doi.org/10.1145/3307681.3325405

faster solutions. Current storage systems address interoperability (i.e., cross-storage system data access) by adding various connectors, such as IBM's Spectrum Scale HDFS Transparency [30] and Intel's Hadoop Adapter [31], and/or middleware libraries, such as IRIS [37] and Alluxio [40]. Nevertheless, better system support is needed for in-transit, in-situ analysis, with scheduling being a big challenge and node sharing impossible in existing solutions [51]. On the other hand, High-Performance Data Analytics (HPDA) [32], the new generation of Big Data applications, involve sufficient data volumes and algorithmic complexity to require HPC resources. For example, Paypal, an online financial transaction platform, and the US Postal Service are using HPC resources to perform fraud detection in real time on billions of transactions and mail scans. Gaining insights from massive datasets while data is being produced by large-scale simulations can enhance the scalability and flexibility of exascale systems [74].

To address this divergence in storage architectures and workload requirements, we have developed LABIOS, a new, distributed, scalable, and adaptive I/O System. LABIOS, a new class of a storage system, is the first (data) LAbel-Based I/O System, is fully decoupled, and is intended to grow in the intersection of HPC and BigData. LABIOS demonstrates the following contributions:

(1) the effectiveness of **storage malleability**, where resources can automatically grow/shrink based on the workload.
(2) how to effectively support **synchronous and asynchronous I/O** with configurable heterogeneous storage.
(3) how to leverage **resource heterogeneity** under a single platform to achieve application and system-admin goals.
(4) the effectiveness of **data provisioning**, enabling in-situ data analytics and process-to-process data sharing.
(5) how to support a diverse set of conflicting I/O workloads, from HPC to BigData analytics, on a single platform, through managed **storage bridging**.

LABIOS achieves these contributions by transforming all I/O requests each into a configurable unit called a *Label*, which is a tuple of an operation and a pointer to data. Labels are pushed from the application to a distributed queue served by a label dispatcher. LABIOS workers (i.e., storage servers) execute labels independently. LABIOS architecture is fully decoupled and distributed. Using labels, LABIOS can offer software-defined storage services and QoS guarantees for a variety of workloads on different storage architectures.

2 BACKGROUND AND MOTIVATION

2.1 Parallel and Distributed File Systems

Parallel file systems (PFS) are the dominant storage solution in most large-scale machines such as supercomputers and HPC clusters and are therefore well understood in the storage community. PFS obey the POSIX standard to offer portable guarantees and strong data consistency. PFS manipulate data in a certain sequence of operations, a paradigm known as *streamlined I/O* (i.e., Unix Standard I/O Streams). Parallel access is achieved by shared file handlers and a complex system of locking mechanisms. Through the years, PFS have been optimized to fit the needs of typical HPC workloads. Application development and storage system design have grown in harmony with one driving the other since the HPC ecosystem is relatively closed to external influence. However, PFS face many limitations [29]. Some relevant to this study include:

a) Storage malleability. Existing high-performance storage solutions are not elastic but static and cannot support power-capped I/O and tunable concurrency control (i.e., QoS guarantees based on job size, priority, input, output, etc.). Sudden workload variations (i.e., I/O demand fluctuations) in distributed systems can be addressed by resource malleability. By dynamically increasing or decreasing the amount of storage resources allocated to an application, the system can reduce its idle resources and therefore achieve lower energy consumption and costs for the end user.
b) Resource utilization. Storage resources are provisioned for the worst-case scenario where multiple jobs happen to enter their I/O-dominant phases simultaneously leading to over/under-provisioning. This issue is worsened by the growing need to support storage resources sharing across multiple clusters via global mounts. Furthermore, allocation exclusivity and over-provisioning due to ignorance or malicious intent also contribute to erroneous resource utilization.
c) Hardware heterogeneity. New storage devices (e.g., SSD, NVMe, etc.,) are being incorporated into system designs resulting in a diverse heterogeneous storage environment. Existing solutions cannot handle this heterogeneity since they assume homogeneous servers. Currently, the responsibility for orchestrating data movement, placement, as well as layout within and across nodes falls on both system administrators and users [47].
d) Flexible interface. Currently, storage is tightly-coupled to certain vendor-specific APIs and interfaces. Even though this ensures consistency and reliability of the storage system, it can also lead to reduced productivity; developers either need to learn new APIs, which limits flexibility, or, adopt new storage systems which leads to environment isolation. Many PFS have introduced various connectors to increase interoperability, but at the cost of performance. Moreover, existing storage systems provide limited facilities for developers to express intent in the form of I/O requirements, semantics, and performance guarantees. Consequently, to achieve good I/O performance, the level of abstraction has been raised. I/O libraries, such as HDF5 [21] and PnetCDF [42] help alleviate this issue but they also add overheads and increase the complexity of use.

In cloud environments the storage scene is different. Innovation is driven by the wide popularity of computing frameworks. As a result, the cloud community has developed a wide variety of storage solutions tailored to serve specific purposes. The most popular storage solution in deployment is the Hadoop Distributed File System (HDFS), which also follows the streamlined I/O paradigm. The architecture and data distribution are somewhat similar to PFS. In HDFS, there are metadata nodes (i.e., namenodes), that are responsible to maintain the namespace, and data nodes that hold the files. However, it has relaxed the POSIX standard to achieve scalability. As the Hadoop ecosystem grows, so are the storage solutions around it: Hive [63] puts a partial SQL interface on front of Hadoop, Pig [52] enables a scripting language in top of MapReduce, HBase [12] applies a partial columnar scheme on top of Hadoop, and HCatalog [23] introduces a metadata layer to simplify access to data stored in Hadoop. This diversity can offer advantages but also undoubtedly increases the complexity of storage. Some of the above solutions suffer from similar limitations as PFS [68], some others are missing critical features [39], and in general most of them perform well for the purpose they were designed for.

2.2 Applications' I/O requirements

Every computing framework expects specific I/O requirements and features from the underlying storage system. Scientific computing for instance, relies mostly on MPI for its computations - communications and domain scientists expect POSIX, MPI-IO, and other high-level I/O libraries to cover their I/O needs. The existing collection of storage interfaces, tools, middleware libraries, data formats, and APIs is deeply instilled in the community and has created a certain mindset of what to expect from the storage stack. Table 1 shows some I/O requirements and how each storage camp, HPC and Cloud, handles them. It also presents proposed optimizations in the literature. Due to those different I/O requirements, there is no "one storage system for all" approach. This is more evident in large scale computing sites, where distributed storage solutions support multiple concurrent applications with conflicting requirements. We believe that future storage systems need a major re-design to efficiently support the diversity of workloads and the explosion of scale.

Feature	I/O requirement	HPC	Cloud	Optimizations
Data consistency	Data passed to the I/O system must be consistent between operations.	Strong, POSIX	Eventual, Immutable	Tunable consistency [65]
File access	Multiple processes must be able to operate on the same file concurrently	Shared Concurrent	Multiple replicas	Collective I/O [62], Concurrent file handlers [22], Complex locks [17, 70, 73]
Global namespace	Data identifiers must be resolved and recognizable in a global namespace that can be accessed from anywhere	Hierarchical Directory, Nesting	Flat	Namespace partitioning [69], Client-side caching [18], Decouple data-metadata [57, 75], File connectors [4, 5, 60]
Fault tolerance	Data must be protected against faults and errors	Specialized hardware, Check-pointing	Data replication, Data partitioning	Erasure coding [67]
Scale	Support for extreme scales and multi-tenancy	Few large jobs, Batch processing	Many small jobs, Iterative	Job scheduling, I/O buffering, Scale-out
Locality	Jobs are spawned where data is	Remote storage	Node local	Data aggregations
Ease of use	Interface, user-friendliness and ease of deployment	High-level I/O libraries	Simple data formats	Amazon S3, Openstack Swift

Table 1: Application I/O Requirements

We provide some examples of workloads that demonstrate the growing need of a storage system that supports diverse workloads on the same single platform.

MOTIVATING EXAMPLES OF I/O WORKLOADS:

a) CM1 (final output, write-intensive): CM1 is a multi-dimensional, non-linear, numerical model designed for idealized studies of atmospheric phenomena [9]. CM1's I/O workload demonstrates a sequential write pattern. The simulation periodically writes collectively its results (e.g., atmospheric points with a set of features) using MPI-IO. Data are written in a binary GrADS format with a shared file access pattern. This workload requires persistence, fault-tolerance, and highly concurrent file access.

b) HACC (check-pointing, update-intensive): HACC stands for Hardware Accelerated Cosmology Code and is a cosmological simulation that studies the formation of structure in collision-less fluids under the influence of gravity in an expanding universe. Each process in HACC periodically saves the state of the simulation along with the dataset using POSIX and a file-per-process pattern. Since HACC runs in time steps, only the last step checkpoint data is needed. Thus, the I/O workload demonstrates an update-heavy pattern. A major performance improvement in HACC workflow is the addition of burst buffers that absorb the checkpointing data faster and perform the last flush of data to the remote PFS.

c) Montage (data sharing, mixed read/write): Montage is a collection of programs comprising an astronomical image mosaic engine. Each phase of building the mosaic takes an input from the previous phase and outputs intermediate data to the next one. It is an MPI-based engine and therefore Montage's workflow is highly dependent on the data migration between processes. The exchange of data between executables is performed by sharing temporary files in the Flexible Image Transport System (FITS) format via the storage system. At the end a final result is persisted as the final jpeg image. The I/O workload consists of both read and write operations using either POSIX or MPI independent I/O.

d) K-means clustering (node-local, read-intensive): This application is a typical and widely used BigData kernel that iteratively groups datapoints into disjoint sets. The input datapoints can be numerical, nodes in a graph, or set of objects (e.g., images, tweets, etc.,). Implementations using the MapReduce framework [15] remain the most popular clustering algorithm because of the simplicity and performance. The algorithm reads the input dataset in phases and each node computes a set of means, broadcasts them to all machines in the cluster and repeats until convergence. The I/O workload is read-intensive and is performed on data residing on the node locally. K-means clustering is typically I/O bound [53].

3 LABIOS

In this section we present the design, architecture, and implementation of LABIOS, a new class of a storage system that uses data-labeling to address the issues discussed in Section 2. LABIOS is a distributed, fully decoupled, and adaptive I/O platform that is intended to grow in the intersection of HPC and BigData.

3.1 Design Requirements

As any distributed storage system, LABIOS is designed to be responsible for the organization, storage, retrieval, sharing, and protection of data. LABIOS also contains a representation of the data itself and methods for accessing it (e.g., read/write). LABIOS' main objective is to *support a wide variety of conflicting I/O workloads under a single platform.* LABIOS is designed with the following principles in mind:

- ***Storage Malleability.*** Applications' I/O behavior consists of a collection of I/O bursts. Not all I/O bursts are the same in terms of volume, intensity, and velocity. The storage system should be able to tune the I/O performance by dynamically allocating/deallocating storage resources across and within applications, a feature called data access concurrency control. Storage elasticity enables power-capped I/O, where storage resources can be suspended or shutdown to save energy. Much like modern operating systems shut down the hard drive when not in use, distributed storage solutions should suspend servers when there is no I/O activity.

- ***I/O Asynchronicity.*** A fully decoupled architecture can offer the desired agility and move I/O operations from the existing streamlined paradigm to a data-labeling one. In data-intensive computing where I/O operations are expected to take a large amount of time,

asynchronous I/O and the data-labeling paradigm are a good way to optimize processing efficiency and storage throughput / latency.

- ***Resource Heterogeneity.*** The hardware composition of the underlying storage should be managed by a single I/O platform. In other words, heterogeneity in hardware (RAM, NVMe, SSD, HDD) but also the presence of multiple layers of storage (i.e., local file systems, shared burst buffers, or remote PFS) should be transparent to the end user. The storage infrastructure should be able to dynamically reconfigure itself to meet the I/O demand of running applications and their I/O requirements. Moreover, storage Quality of Service (QoS) guarantees are a highly desired feature that can be achieved by efficiently matching the supply to the I/O demand [46].

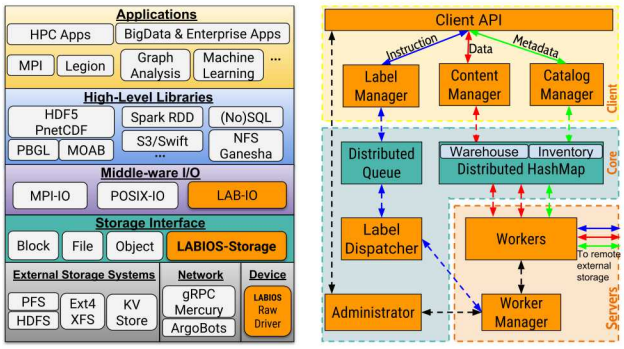
-***Data provisioning.*** The I/O system should be programmable (i.e., policy-based provisioning and management). Storage must naturally carry out data-centric architectures (e.g., ActiveStorage [58], or ActiveFlash [64]), where data operations can be offloaded to the storage servers relieving the compute nodes of work such as performing data filtering, compression, visualization, deduplication, or calculating statistics (i.e., Software Defined Storage (SDS)). Offloading computation directly to storage and efficient process-to-process data sharing can significantly reduce expensive data movements and is the pinnacle of success for data-centric architectures [54].

- ***Storage Bridging.*** The I/O system should abstract low-level storage interfaces and support multiple high-level APIs. Modern distributed computing makes use of a variety of storage interfaces ranging from POSIX files to REST objects. Moreover, existing datasets are stored in a universe of storage systems, such as Lustre, HDFS, or Hive. Storage solutions should offer developers the ability to use APIs interchangeably avoiding interface isolation and, thus, boost user productivity while minimizing programmability errors.

3.2 Architecture

3.2.1 Data model. The core of LABIOS storage is a ***Label***, which is effectively a tuple of one or more operations to perform and a pointer to its input data. It resembles a shipping label on top of a Post Office package where information such as source, destination, weight, priority, etc., clearly describe the contents of the package and what should happen to it. LABIOS' label structure includes: type, uniqueID, source and destination as pointers (i.e., can be a memory pointer, a file path, a server IP, or a network port), operation to be performed as a function pointer (i.e., all functions, user- or pre-defined, are stored in a shared program repository which servers have access to), and a collection of flags indicating the label's state (i.e., queued, scheduled, pending, cached, invalidated, etc.). In essence, labels encapsulate the instructions to be executed on a piece of data. All I/O operations (e.g., fread() or get(), fwrite() or put(), etc.,) are expressed in the form of one or more labels and a scheduling policy to distribute them to the servers. Labels belong to each application exclusively. They are immutable, independent of one another, cannot be reused, and can be executed by any worker anywhere in the cluster. In contrast, labels are not a computation decomposition (i.e., compute task), or a simple data object encapsulation (e.g., RDDs) but rather a storage-independent abstraction that simply expresses the application's intent to operate on certain data.

3.2.2 Overview. As it can be seen in Figure 1(a), LABIOS can be used either as a middleware I/O library or as a full stack storage solution. Applications can use the LABIOS library to perform I/O using labels and take advantage of the full potential of the system.



(a) Software Stack (b) Architecture - Components

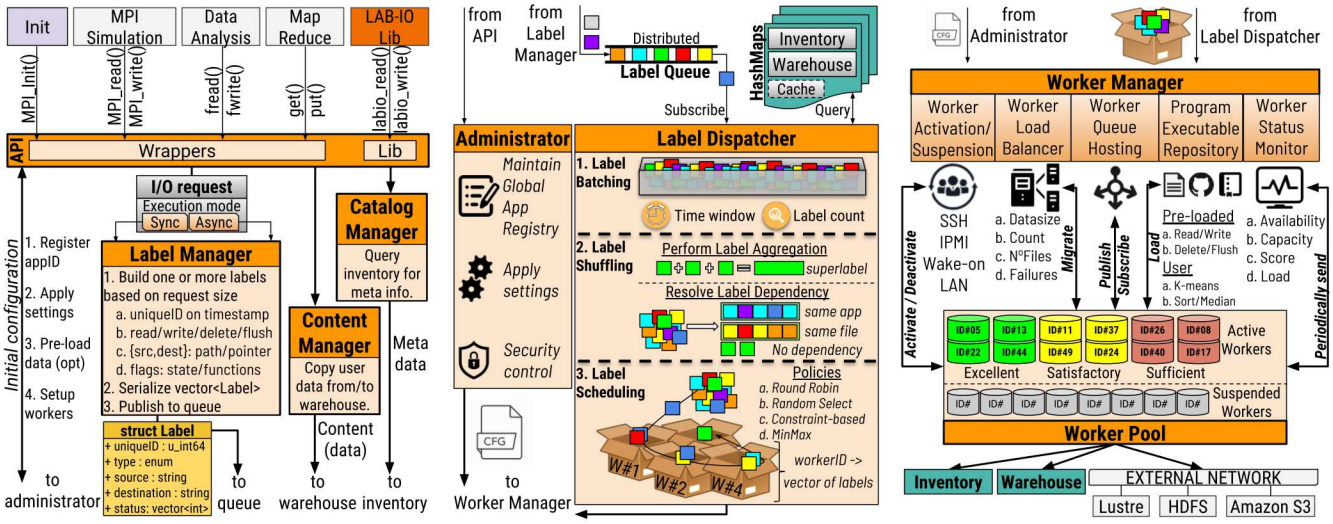
Figure 1: LABIOS overview.

Each label can carry a set of functions to be performed by the storage server that executes it. For instance, an application can push write labels and instruct LABIOS to first deduplicate entries, sort the data, compress them, and finally write them to the disk. On the other hand, to maintain compatibility with existing systems, legacy applications can keep their I/O stack and issue typical I/O calls (e.g., fwrite()). LABIOS will intercept those I/O calls, transform them into labels, and forward them to the storage servers. LABIOS can also access data via its own raw driver that handles data on the storage device in the form of labels. By adding more servers, the capacity and performance of them is aggregated in a single namespace. Furthermore, LABIOS can unify multiple namespaces by connecting to external storage systems, a feature that allows LABIOS to offer effective *storage bridging*.

LABIOS offers high speed data access to parallel applications by splitting the data, metadata, and instruction paths and decoupling storage servers from the application, as shown in Figure 1(b). This decoupling of clients and servers is a major architectural choice that enables several key features in LABIOS: the power the *asynchronous I/O*, the effectiveness of *data provisioning*, and the proliferation of *heterogeneous storage* resources. An incoming application first registers with LABIOS, upon initialization, and, passes workload-specific configurations to set up the environment. LABIOS receives the application's I/O requests via the client API, transforms them, using the label manager, into one or more labels depending mostly on the request size, and then pushes them into a distributed label queue. Users' data are passed to a distributed data warehouse and a metadata entry is created in an inventory. A *label dispatcher* consumes the label queue and distributes labels using several scheduling policies. Storage servers, called LABIOS *workers*, are organized into a worker pool with a manager being responsible to maintain its state. Workers can be suspended depending on the load of the queue creating an elastic storage system that is able to react to the state of the cluster. Lastly, workers execute their assigned labels independently and read/write data either on their own storage device or through a connection to an external storage system.

3.2.3 Components. LABIOS consists of three main components connected and configured together as shown in Figure 2:

1. **LABIOS Client**, subfigure 2(a): This component interacts with the application and has three main goals: a) per-application system initialization: register application info (i.e., ID, group name, group



(a) **Client** (b) **Core** (c) **Server**

Figure 2: LABIOS internal design.

credentials and permissions), apply application-specific settings, pre-load data from external sources (if needed), and setup LABIOS workers. b) accept application's I/O requests, either by intercepting existing I/O calls using function call wrappers or by exposing LABIOS API, and, c) build labels based on the incoming I/O request.

Label Manager: builds one or more labels based on the request characteristics (e.g., read/write, size, file path, etc.,), serializes and publishes them to the distributed label queue. Each label gets a unique identifier based on the origin of the operation and a timestamp (in nanoseconds), which ensures the order of operations (i.e., this is the constraint in the priority queue). Labels are created by a configurable size parameter within a range of min and max values (e.g., min 64KB - max 4MB). The data size parameter in each label is the unit of data distribution in the system. An I/O request larger than the maximum label size will be split into more labels creating a 1-to-N relationship between request and number of labels (e.g., for a 10MB `fwrite()` and 1MB max label size, 10 labels will be created). Any I/O request smaller than the minimum label size will be cached and later aggregated in a special indexed label to create a N-to-1 relationship between number of requests and label (e.g., for ten 100KB `fwrite()` and 1MB max label size, one label will be created). Lastly, these thresholds can be bypassed for certain operations, mostly for synchronous reads. Setting min and max label size values is dependent on many system parameters such as memory page size, cache size, network type (e.g., TCP buffer size), and type of destination storage (e.g., HDDs, NVMe, SSDs). LABIOS can be configured in a *synchronous* mode, where the application waits for the completion of the label, and in *asynchronous* mode, where the application pushes labels to the system and goes back to computations. A waiting mechanism, much like a barrier, can be used to check the completion of a single or a collection of asynchronously issued labels. The async mode can significantly improve the system's throughput but it also increases the complexity of data consistency and fault tolerance guarantees.

Content Manager: is mainly responsible for handling user data inside a *warehouse*.The warehouse is implemented by a distributed hashmap (i.e., key-value store), it temporarily holds data in-memory effectively serving as the bridge between clients and workers. The warehouse is a collection of system-level structures (i.e., tables in the distributed key-value store), that are application-specific, and has the following requirements: highly available, concurrent data access, fault tolerant, and high throughput. The content manager exposes the warehouse via a simple get/put/delete interface to both the clients and the workers. The size and location of the warehouse is configurable based on several parameters such as number of running applications, application's job size, dataset aggregate size, and number of nodes (e.g., one hashtable per node, or per application). Every entry in the warehouse is uniquely identified by a key which is associated with one or more labels. The content manager can also create ephemeral regions of the warehouse (e.g., temporary rooms) which can be used for workflows where data are shared between processes (i.e., data sharing, Section 2.2). Data flows through LABIOS as follows: from application's buffer to the warehouse, and from there to worker storage for persistence or to another application's buffer. Lastly, the content manager also provides a cache to optimize small size data access; a known issue in distributed storage [10]. I/O requests smaller than a given threshold are kept in a cache and, once aggregated, a special label is created and pushed to the distributed queue to be scheduled to a worker (much like memtables and SSTables in LevelDB). This minimizes network traffic and can boost the performance of the system.

Catalog Manager: is responsible to maintain both user and system metadata information in an inventory, implemented by a distributed hashmap. The catalog manager exposes an interface for each application to query and update the entries within the inventory. Decentralization of the catalog services makes the system scalable and robust. Multiple concurrent processes can query the inventory at the same time. For concurrent updates, LABIOS

adopts the semantics of the underlying distributed hashmap with high-availability and concurrent access ensuring the correctness and high throughput of catalog operations. LABIOS also offers the flexibility to place the inventory in memory for high performance, protected by triple replication for fault tolerance. However, this increases the memory footprint of LABIOS and it depends on the availability of resources. The organization of inventory entries depends on the data model (files, objects, etc.) and/or high-level I/O libraries and middleware. For instance, for POSIX files the inventory entries may include: filename to file stat, file handler to filename, file handler to file position in offset, filename to a collection of labels, and others. An HDF5 or a JSON file will have different inventory entries. LABIOS-specific catalog information include: label status (e.g., in-transit, scheduled, pending), label distribution (e.g., label to workerID), label attributes (e.g., ownership, flags), and location mappings between user's data and LABIOS internal data structures (e.g., a user's POSIX file might be stored internally as a collection of objects residing in several workers). Lastly, when LABIOS is connected to external storage resources, it relies on their metadata service. LABIOS becomes a client to the external storage resources and "pings" their metadata service to acquire needed information. LABIOS does not keep a copy of their respective metadata internally to avoid possible inconsistent states. However, further investigation is needed to optimize this process by avoiding added network latencies from external sources.

2. **LABIOS Core**, subfigure 2(b): This component is responsible to manage the instruction, data, and metadata flow separately.

Administrator: maintains the system's state by keeping track of all running applications in a global registry, setting up the environment per application (e.g., boot up exclusive workers if needed, pre-load data from external sources, etc.), and performing security control via user authentication and access permission checks.

Label Queue: LABIOS distributed queuing system has the following requirements: high message throughput, always on and available, at-most-once delivery guarantees, highly concurrent, and fault tolerant. These features ensure data consistency since the label dispatcher will consume labels once and in order. The queue concurrency ensures that multiple dispatchers can service the same queue or one dispatcher multiple queues. The number of queues is configurable based on the load (e.g., one queue per application, or one queue per 128 processes, or one queue per node).

Label Dispatcher: subscribes to one or more distributed label queues and dispatches labels to workers using several scheduling policies. The label dispatcher is multi-threaded and can run on one or more nodes depending on the size of the cluster. LABIOS dispatches labels based on either a time window or the number of labels in the queue; both of those parameters are configurable. For example, the dispatcher can be configured to distribute labels one by one or in batches (e.g., every 1000 labels). To avoid stagnation, a timer is also used; if the timer expires, LABIOS will dispatch all available labels in the queue. Furthermore, the number of label dispatchers is dynamic and depends on the number of deployed queues. There is a fine balance between the volume and velocity of label production stemming from the applications and the rate at which the dispatcher handles them. The relationship between the dispatcher and queuing system increases the flexibility and scalability of the platform and provides an infrastructure to match the rate of incoming I/O. The dispatcher consists of two phases:
a) *Label Shuffling*: takes a vector of labels as an input and shuffles them based on type and flags. Two operations are performed by the shuffler. *Data aggregation*: labels that reflect user's requests in consecutive offsets can be combined to one larger label to maintain locality (this feature can be turned on or off). *Label dependencies*: data consistency must be preserved for dependent labels. For instance, a read after write pattern; LABIOS will not schedule a read label before the dependent write label completes. To resolve such dependencies, the shuffler will create a special label, called *supertask*, which embodies a collection of labels that need to be executed in strictly increasing order. After sorting the labels and resolving dependencies, the shuffler sends labels either to the solver to get a scheduling scheme, or directly to the assigner depending on the type (e.g., a read label is preferably assigned to the worker that holds the data to minimize worker-to-worker communication).
b) *Label Scheduling*: takes a vector of labels as an input and produces a dispatching plan. For a given set of labels and workers, the scheduler answers three main questions: how many workers are needed, which specific workers, and which labels are assigned to which workers. LABIOS is equipped with several scheduling policies (in detail in Section 3.3). A map of {workerID, vector of labels} is passed to the worker manager to complete the assignment by publishing the labels to each individual worker queue. Labels are published in parallel using a thread pool. The number of threads in the pool depends on the machine the label dispatcher is running on as well as the total number of available workers.

3. **LABIOS Server**, subfigure 2(c): This component is responsible for managing the storage servers and has two main entities:

Worker: is essentially the storage server in LABIOS. It is fully decoupled from the applications, is multithreaded, and runs independently. Its main responsibilities are: a) service its own queue, b) execute labels, c) calculate its own worker score and communicate it to the worker manager, d) auto-suspend itself if there are no labels in its queue for a given time threshold, and e) connect to external storage sources. The worker score is a new metric, critical to LABIOS operations, that encapsulates several characteristics of the worker into one value which can then be used by the label dispatcher to assign any label to any appropriate worker. A higher scored worker is expected to complete the label faster and more efficiently. The score is calculated by every worker independently at an interval or if substantial change of status occurs, and it includes: i) *availability*: 0 not-available (i.e., suspended or busy), 1 available (i.e., active and ready to accept labels). ii) *capacity*: (double) [0,1] based on the ratio between remaining and total capacity. iii) *load*: (double) [0,1] based on the ratio between worker's current queue size and max queue size (the max value is configurable). iv) *speed*: (integer) [1,5] based on maximum bandwidth of worker's storage medium and grouped based on ranges (e.g., 1: ≤200MB/s, 2: 200-550MB/s, ... 5: ≥3500MB/s). v) *energy*: (integer) [1,5] based on worker's power wattage on full load (e.g., an ARM-based server with flash storage consumes less energy than a Xeon-based server with a spinning HDD). The first three are dynamically changing based on the state of the system whereas speed and energy variables are set during initialization and remain static. Lastly, each variable is multiplied by a weight. LABIOS' weighting system is set in place

to express the scheduling policy prioritized (examples shown in Table 2)). For instance, if energy consumption is the constraint that the label dispatcher aims to optimize then the energy variable gets a higher weight. The final score is a float in range between 0 and 1 and is calculated as: $Score_{worker(i)} = \sum_{j=1}^{5} Weight_j * Variable_j$.

Priority	Availability	Capacity	Load	Speed	Energy
Low latency	0.5	0	0.35	0.15	0
Energy savings	0	0.15	0.2	0.15	0.5
High Bandwidth	0	0.15	0.15	0.70	0

Table 2: LABIOS worker's score - Weighting examples.

Worker Manager: is responsible for managing the worker pool. Its responsibilities are: a) maintain workers' statuses (e.g., remaining capacity, load, state, and score) in a distributed hashmap (in-memory or on disk), b) host the workers' queues, c) perform load balancing between workers, and d) dynamically commission/decommission workers to the pool. It is connected to the administrator for accepting initial configurations for incoming applications, and to the label dispatcher for publishing labels in each worker's queue. It can be executed independently on its own node by static assignment, or dynamically on one of the worker nodes by election among workers. In a sense, the worker manager partially implements objectives similar to other cluster resource management tools such as Zookeeper, or Google's Borg. One of the most performance-critical goals of the worker manager is to maintain a sorted list of workers based on their score. Workers update their scores constantly, independently, and in a non-deterministic fashion. Therefore, the challenge is to be able to quickly sort the updated scores without decreasing the responsiveness of the worker manager. LABIOS addresses this issue by a custom sorting solution based on buckets. The set of workers are divided on a number of buckets (e.g., high, medium, and low scored workers) and an approximate bin sorting algorithm is applied [25]. A worker score update will only affect a small number of buckets and the complexity time is relevant to the size of the bucket. Lastly, the worker manager can send activation messages to suspended workers either by using the administrative network, if it exists, (i.e., `ipmitool --power on`), or by a custom solution based on ssh connections and wake-on-lan tools.

3.2.4 Deployment Models. The power and potential of LABIOS' flexible and decoupled architecture can be seen in the several ways the system can be deployed. Depending on the targeted hardware and the availability of storage resources, LABIOS can: a) replace an existing parallel or distributed storage solution, or b) be deployed in conjunction with one or more underlying storage resources as an I/O accelerator (e.g., burst buffer software, I/O forwarding, or software-defined storage in user space). Leveraging the latest trends in hardware innovation, the machine model we present here as our basis for several deployment schemes is as follows: compute nodes equipped with a large amount of RAM and local NVMe devices, an I/O forwarding layer [33], a shared burst buffer installation based on SSD equipped nodes, and a remote PFS installation based on HDDs (motivated by the recent machines Summit in ORNL or Cori on LBNL). Figure 3 shows four equally appropriate deployment examples that can cover different workloads:
(a) *LABIOS as I/O accelerator* (fig. 3(a)): can be used as a fast distributed cache for temporary I/O or on top of other external sources. It is also ideal for Hadoop workloads with node-local I/O. However,

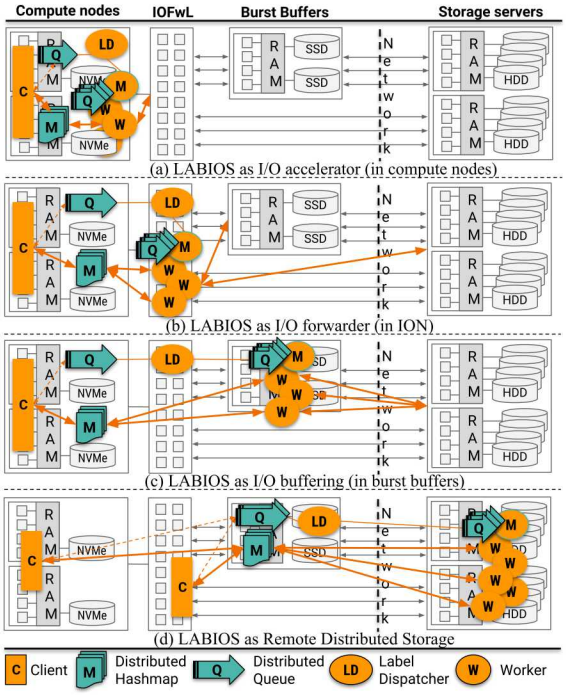


Figure 3: LABIOS deployment schemes.

it must use some compute cores to run its services and I/O traffic will mix with the compute network.
(b) *LABIOS as I/O forwarder* (fig. 3(b)): ideal for asynchronous I/O calls where applications pass their data to LABIOS which pushes them in a non-blocking fashion to remote storage, either native to the system or external. However, its scalability is limited by the size of the I/O forwarding layer.
(c) *LABIOS as I/O buffering* (fig. 3(c)): ideal for fast temporary storage, data sharing between applications, and in-situ visualization and analysis. Requires additional storage and network resources.
(d) *LABIOS as remote distributed storage* (fig. 3(d)): offers better system scalability by scaling each individual component independently, better resource utilization, and higher flexibility to the system administrator. For instance, one can increase the number of client queues in scenarios when label production is high, or deploy more dispatchers to distribute labels faster. It has, however, higher deployment complexity. LABIOS' fully decoupled architecture provides greater flexibility and promotes scalability; I/O scales along with the application by simply provisioning additional resources.

3.3 Implementation

Label Scheduling: LABIOS balances the rate of incoming labels, the dispatching cost, and the time to execute the labels and offers an flexible, intent-aware infrastructure. LABIOS provides a custom data distribution by implementing different scheduling policies:
a) *Round Robin*: given a set of labels and a list of available workers the dispatcher will distribute labels in a round robin fashion, much like a PFS does. The responsibility of activating workers and compiling a list of available workers for every scheduling window falls under worker manager. This policy demonstrates low scheduling cost but additional load balancing between workers might occur.

```
#include <tabios.hpp>
...
Client client = InitClient(ip, port, connConfig);
std::string path = "pvfs2:/data/integers.dat";
LabelSrc src = new LabelSrc(path, src_offset, size);
LabelType type = SDS_IN_SITU; //complex type
LabelFlag flags = CACHED | MPI_IO; //keep in cache
std::function<int(vector<int>)> fn = FindMedian;
Label label = client.CreateLabel(type,src,fn,flags);
Status status =client.IPublishLabel(label);
...                    //perform other computations
client.WaitLabel(&status);
int median = std::static_cast<int>(status.data);
```

Figure 4: LABIOS API example.

b) *Random Select*: given a set of labels, the dispatcher will distribute labels to all workers randomly regardless of their state (i.e., active or suspended). This policy ensures the uniform distribution of workload between workers, low scheduling cost, but with no performance guarantees (i.e., possible latency penalty by activating suspended workers, or lack of remaining capacity of worker, etc.).
c) *Constraint-based*: in this policy, LABIOS provides the flexibility to express certain priorities on the system. Through the weighting system of worker scores, discussed in Section 3.2.3, the dispatcher will distribute labels to workers based on the constraint with higher weight value. The constraints used are: a) *availability*, active workers will have higher score. b) *worker load*, based on worker's queue size. c) *worker capacity*, based on worker's remaining capacity. d) *performance*, workers with higher bandwidth and lower latency get higher score. For a given set of labels, the dispatcher requests a number of workers with the highest score, respective to the prioritized constraint, from the worker manager and distributes the labels evenly among them. The number of workers needed per a set of labels is automatically determined by LABIOS based on the total aggregate I/O size and the selected constraint balancing parallel performance and efficiency. These heuristics can be configured and further optimized based on the workload.
d) *MinMax*: given a set of labels and a collection of workers, the dispatcher aims to find a label assignment that maximizes I/O performance while minimizing the system's energy consumption, subject to the remaining capacity and load of the workers; essentially a minmax multidimensional knapsack problem, a well-known NP-hard combinatorial optimization problem [55]. LABIOS solves this problem using an approximate dynamic programming (DP) algorithm [3] which optimizes all constraints from the previous policy. This policy gives a near-optimal matching of labels - workers but with a higher scheduling cost.
API: LABIOS exposes a label API to the application to interact with data. The storage interface expresses I/O operations in the form of labels. The API includes calls to create-delete, publish-subscribe labels, among others. LABIOS' API offers higher flexibility and enables software defined storage capabilities. For instance, the code snippet shown in Figure 4, creates an asynchronous label which reads a file that includes a collection of integers from an external PFS using the MPI-IO driver, calculates the median value, and passes only the result back to the application via asynchronous I/O.
LABIOS Prototype Implementation Details: LABIOS is written in C++ and has approximately 10K lines of code. We also use several external open source projects in our LABIOS prototype. For the distributed queuing system, used widely in LABIOS both in client and workers, we used NATS server [14], a simple, high-performance open source messaging system. NATS was selected due to its superiority in throughput (i.e., >200K msg/sec, 12x higher than Apache ActiveMQ and 3x than Kafka), low latency, and light-weight nature. For the distributed hashmaps, we used a custom version of Memcached with the *extstore* [49] plugin. Memcached is a simple in-memory key-value store, with easy deployment, development, and great APIs. The extstore plugin allows us to store memcached data to a storage device (e.g., NVMe, SSD) instead of RAM. We also modified the default key distribution from randomly hashing keys to servers, to a node local scheme to increase throughput and promote locality. Effectively, each node in LABIOS stores its hashtables on its local memcached daemon. For label serialization, we used Cereal [27], a header-only binary serialization library which is designed to be fast, light-weight, and easy to extend. For hashing we used Citihash algorithms [26], for memory allocations TCMalloc, and lastly, for metadata indexing an in-memory B-Tree implementation by Google.

3.4 Discussion

Considerations: LABIOS' design and architecture promotes its main objective of supporting a diverse variety of conflicting I/O workloads under a single platform. However, additional features could be derived from LABIOS label paradigm. *Fault tolerance*: In the traditional streamlined I/O paradigm, if an fwrite() call fails the entire application fails and it must restart to recover (i.e., using check-pointing mechanisms developed especially in the scientific community). LABIOS' label granularity and decoupled architecture could provide the ability to repeat a failed label and allow the application to continue without restarting. *Energy-awareness*: First, LABIOS' ability to dynamically commission/decommission workers to the pool creates an elastic storage solution with tunable performance and concurrency control but also offers a platform that could leverage the energy budget available. One could observe the distinct compute-I/O cycles and redirect energy from compute nodes to activate more LABIOS workers for an incoming I/O burst. Second, LABIOS' support of heterogeneous workers could lead to energy-aware scheduling where non mission-critical work would be distributed on low-powered storage nodes, effectively trading performance for power consumption. Lastly, *storage containerization* could be a great fit for LABIOS' decoupled architecture. Workers could execute multiple containers running different storage services. For instance, workers would host one set of containers running Lustre servers and another running MongoDB. The worker manager would act as the container orchestrator and the label dispatcher could manage hybrid workloads by scheduling labels to both services under the same runtime.
Challenges and Limitations: During the design and implementation of LABIOS we have identified several challenges.
Multitenancy and Security: How does LABIOS handle multiple applications accessing the system? Contention avoiding techniques must be employed. LABIOS could handle this by operating in isolation, where each application would have had its own queues, label dispatchers, and workers; an additional layer of global application

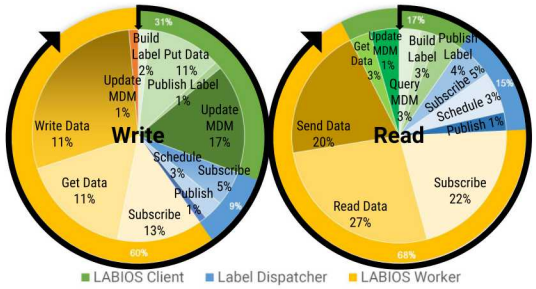


Figure 5: Anatomy of LABIOS operations.

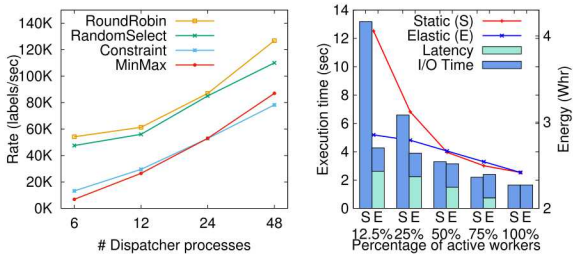
orchestrator [38] could deal with the overall system traffic. Further concerns might arise by multitenancy such as user authentications and security. For instance, the LABIOS prototype uses NATS server as the queuing system. As of now, NATS does not support TLS and SSL which could limit the secure capabilities of the system.

Concurrent operations: There are many components in LABIOS that need to be efficiently accessed at the same time. The distributed queuing system needs to support highly concurrent label insertion. The distributed hashmaps have to support a large number of clients for accessing both data and metadata. For instance, LABIOS' constraint on the priority queue is the timestamp which means clock skewness across the cluster can affect the order of operations in the system. LABIOS' design relies heavily on the characteristics and performance of those components and any limitations from their side could become limitations of the entire LABIOS system.

I/O request decomposition: The main question LABIOS faces is how to transform I/O requests into labels. What would be an optimal decomposition granularity? LABIOS' label manager addresses this by splitting requests based on an I/O size range. For small requests, LABIOS caches them and aggregate them into a larger label. For large requests, LABIOS splits them into more labels offering a higher degree of parallelism. We plan to leverage the underlying hardware characteristics (network buffers, RAM page size, disk blocks, etc.) to refine LABIOS I/O request decomposition strategy. Another question, relevant to label granularity, is how are label dependencies and session/service management handled? Any task-based system faces these challenges [66]. LABIOS resolves label dependencies based on a configurable policy-driven granularity (i.e., per-application, per-file, per-dataset, etc.,). We plan to further LABIOS ability to resolve label dependencies by using dependency graphs.

4 EVALUATION

Methodology: All experiments were conducted on a bare metal configuration offered by Chameleon systems [11]. The total experimental cluster consists of 64 client nodes, 8 burst buffer nodes, and 32 storage servers. Each node has a dual Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz (i.e., a total of 48 cores per node), 128 GB RAM, 10Gbit Ethernet, and a local HDD for the OS. Each burst buffer node has the same internal components but, instead of an HDD, it is equipped with SSDs. The cluster OS is CentOS 7.1, the PFS we used is OrangeFS 2.9.6. In terms of workloads, we used all four applications from Section 2.2 and a synthetic benchmark. LABIOS has been deployed in the cluster as a storage solution (Fig. 3 (d)).



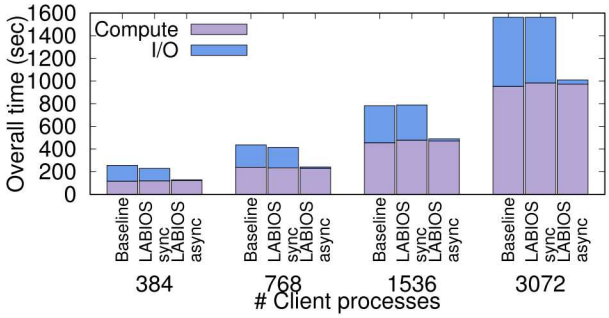
(a) Label Scheduling (b) Storage Malleability.

Figure 6: LABIOS internal component evaluation.

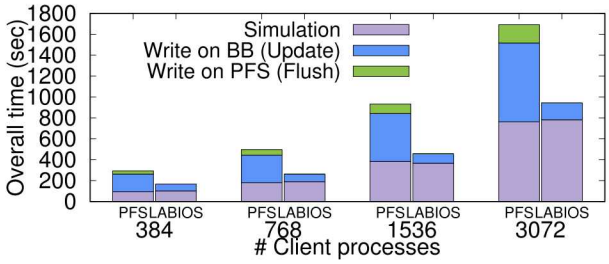
- ***Anatomy of LABIOS read/write operations***: Figure 5 shows decomposition of the read and write label execution expressed as time percentage and divided by each LABIOS component. For instance, a *write label* starts with the LABIOS client building a label (at 12 o'clock on the figure) which takes 2% of the total time, it then passes the data to the warehouse (put data 11%), publishes the label to the queue (1%), and finally updates the catalog manager (MDM) about the operation (17%). The total LABIOS client operations take 31% of the total time. The label journey continues in the label dispatcher who picks up the label from the queue (subscribe 5%), schedules it (3%), and pushes it to a specific worker's queue (publish 1%). The most work is done by the LABIOS worker (60% of the total operation time) who first picks up the label from its queue and the data from the warehouse (get data 17%), writes the data down to the disk (29%), and finally updates the catalog manager (1%). All results are the average time of executing a 1MB label 10K times.

- ***Label dispatching***: in this test, we present how LABIOS performs with different scheduling policies and by scaling the number of label dispatcher processes. We report the rate (i.e., labels per second) at which each scheduling policy handles incoming labels. LABIOS client runs on all 64 client machines, the label dispatcher is deployed on its own dedicated node, and LABIOS workers run on the 32 server machines. We measure the time the dispatcher takes to distribute 100K randomly generated labels (i.e., mixed read and write equally sized labels). As it can be seen in Figure 6(a), all policies scale linearly as we scale the label dispatcher processes from 6-48 (i.e., equal to max cores of the node). Round-robin and random-select achieve comparable scheduling rates between 55-125K labels per second. Constraint-based is more communication intensive since it requires exchanging information about the workers with their manager. MinMax scales better with more resources since it is more CPU intensive (i.e., DP approach).

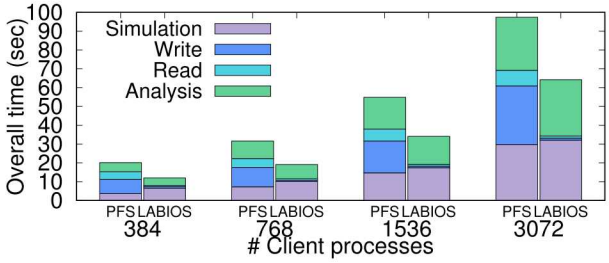
- ***Storage malleability***: in this test, we present how LABIOS elastic storage feature affects I/O performance and energy consumption. We issue 4096 write labels of 1MB each and we measure the total I/O time stemming from different ratios between active workers over total workers (e.g., 50% ratio means that 16 workers are active and 16 are suspended). A suspended worker can be activated in about 3 seconds on average (in our testbed between 2.2-4.8 seconds). Figure 6(b) demonstrates the importance of balancing the added latency to activate more workers and the additional performance we get. We show two worker allocation techniques, the static (S), where labels are placed only on the active workers, and the elastic (E), where more workers activate to serve incoming I/O. When LABIOS has



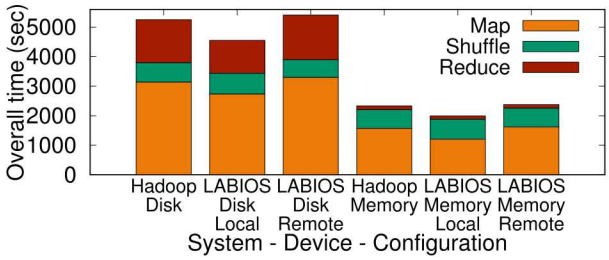
(a) I/O asynchronicity - CM1 performance.



(b) Resource heterogeneity - HACC performance.



(c) Data provisioning - Montage performance.



(d) Storage Bridging - Hadoop K-Means performance.

Figure 7: LABIOS performance evaluation.

a small percentage of active workers, the elastic strategy can boost performance significantly even though we pay the latency penalty to activate more workers. However, when we have a sufficient number of active workers (e.g., 75% or 24 out of 32 total workers), waking up more workers hurts the performance due to the latency penalty. This is further apparent when we see the energy efficiency of the system, expressed in watts per hour (Whr). In our testbed, active workers consume 165 watts, whereas suspended workers only 16 watts. LABIOS elastic worker allocation makes sense until the 75% case where the static allocation is more energy efficient.

- ***I/O asynchronicity***: LABIOS supports both synchronous and asynchronous operations. The potential of a label-based I/O system is more evident by the asynchronous mode where LABIOS can overlap the execution of labels behind other computations. In this test, LABIOS is configured with the round robin scheduling policy, label granularity of 1MB, and the label dispatcher uses all 48 cores of the node. We scaled the clients from 384 to 3072 processes (or MPI ranks in this case) to see how LABIOS scales. We run CM1 in 16 iterations (i.e., time steps) with each step first performing computing and then I/O. Each process is performing 32MB of I/O with the total dataset size reaching 100GB per step for the largest scale of 3072. As it can be seen in Figure 7(a), LABIOS scales well with the synchronous mode, offering competitive performance when compared with our baseline, an OrangeFS deployment using the same number of storage servers (i.e., 32 servers). When LABIOS is configured in the async mode, each I/O phase can be executed overlapped with the computation of the next step. This results in a significant 16x I/O performance boost, and a 40% execution time reduction since the I/O is hidden behind computation. Note that no user code change is required. LABIOS intercepts the I/O calls and builds labels that get executed in a non-blocking fashion.

- ***Resource heterogeneity***: in this test, we run HACC also in 16 time steps. At each step, HACC saves its state on the burst buffers and only at the last step persists the checkpoint data to the remote storage, an OrangeFS deployment. This workload is update-heavy. LABIOS is configured similarly as before but with support of heterogeneous workers, 8 SSD burst buffers and 32 HDD storage servers. LABIOS transparently manages the burst buffers and the servers, and offers 6x I/O performance gains, shown in Figure 7(b). Moreover, worker to worker flushing is performed in the background.

- ***Data provisioning***: in this test, we run Montage, an application that consists of multiple executables that share data between them (i.e., output of one is input to another). LABIOS is configured similarly to the previous set of tests. The baseline uses an OrangeFS deployment of 32 servers. In this test, the simulation produces 50GB of intermediate data that are written to the PFS and then passed, using temporary files, to the analysis kernel which produces the final output. As it can be seen in Figure 7(c), our baseline PFS spends significant time in I/O for this data sharing via the remote storage. This workflow can be significantly boosted by making the data sharing more efficient. LABIOS, instead of sharing intermediate data via the remote storage, passes the labels from the simulation to the analysis via the distributed warehouse. Each intermediate data file creates labels where the destination is not LABIOS workers but the analysis compute nodes. This accelerates the performance in two ways: a) no temporary files are created in the remote storage servers, and b) simulation and analysis can now be pipelined (i.e., analysis can start once the first labels are available). As a result, LABIOS offers 65% shorter execution time, boosts I/O performance by 17x, and scales linearly as the number of clients grow.

- ***Storage Bridging***: Figure 7(d) demonstrates the results of running K-means clustering. Our baseline is a 64-node HDFS cluster. LABIOS is configured in two modes: node-local I/O, similar to the

HDFS cluster, and remote external storage, similar to an HPC cluster (Section 3.2.4 (a) & (d)). In the first mode, LABIOS workers run on each of the 64 nodes in the cluster whereas in the second mode, data resides on an external storage running on 32 separate nodes. This application has three distinct phases: a) *Map*, each mapper reads 32MB from storage, performs computations, and then writes back to the disk 32MB of key-value pairs. b) *Reduce*, each reducer reads 32MB of key-value pairs written from the mappers and performs further computations, c) *Shuffle*, all values across all reducers in the cluster are exchanged via the network (i.e., 32MB network I/O). Finally, it writes the new final centroids back to the disk. An optimized version of this algorithm (i.e., Apache Mahout) avoids writing the key-value pairs back to HDFS during map phase, but instead it emits those values to the reducers avoiding excessive disk I/O (i.e., `Hadoop-Memory` in figure 7(d)). LABIOS supports this workload by having each worker on every node reading the initial dataset in an optimized way by performing aggregations, much like MPI collective-I/O where one process reads from storage and distributes the data to all other processes. Further, LABIOS decoupled architecture allows the system to read data from external resources (i.e., `LABIOS-Disk-Remote` in figure 7(d)). As it can be seen in the results, reading from external sources is slower than the native node-local I/O mode but it is still a feasible configuration under LABIOS, one that leads to the avoidance of any expensive data movements or data-ingestion approach.

5 RELATED WORK

Innovation and new features in modern storage: fixed reservation with performance guarantees in Ceph [69], in-memory ephemeral storage instances in BeeGFS [28], decoupling of data and metadata path in latest versions of OrangeFS [59], and client-to-client coordination with low server-side coupling in Sirocco [16]. Our work is partially inspired by the above developments. LABIOS is able to offer these features due to its innovative design and architecture.
Active storage: Comet [24] an extensible, distributed key-value store that seeks application-specific customization by introducing active storage objects. Comet's design allows storage operations as a result of executing application specific handlers. ActiveFlash [64] an in-situ scientific data analysis approach, wherein data analysis is conducted on where the data already resides. LABIOS workers can independently execute data-intensive operations in a non-blocking fashion since they are fully decoupled from the clients.
Workflow Interoperability: Running data analysis along with computationally challenging simulations has been explored by [7]. Dataspaces [20] offers a semantically specialized virtual shared space abstraction to support multiple interacting processes and data-intensive application workflows. DAOS [8] integrates a high-performance object store into the HPC storage stack and supports a flexible interface for diverse workloads. However, dedicated analysis resources or expensive data movement between different clusters is still required. LABIOS' label describes the destination of a certain data operation and can be a memory buffer or a file on another compute node making data sharing easy and efficient.
Task-based Computation Frameworks: Machine independent parallel task-based computing paradigms with new runtime systems such as Charm++ [34] and Legion [1] have been long advocating for splicing computation to smaller, independent pieces that can be better managed [41], scheduled [43, 45], and executed [48] on heterogeneous environments. LABIOS, in a sense, realizes the same vision of work decomposition but for I/O jobs and not computations.
Storage Malleability: elasticity is a well explored feature in Cloud storage. Dynamic commission of servers in HDFS [13], transactional database properties with elastic data storage such as ElasTraS [19], and several works exploring energy efficiency in storage systems [2, 44]. LABIOS inherits this feature by its decoupled architecture and the worker pool design and brings storage malleability to HPC as well as BigData.

6 CONCLUSIONS AND FUTURE WORK

Modern large-scale storage systems are required to support a wide range of workflows with different, often conflicting, I/O requirements. Current storage solutions cannot definitively address issues stemming from the scale explosion. In this paper, we present the design principles and the architecture of a new, distributed, scalable, elastic, energy-efficient, and fully decoupled label-based I/O system, called LABIOS. We introduce the idea of a *label*, a fundamental piece of LABIOS' architecture, that allows the I/O system to provide storage flexibility, versatility, agility, and malleability. Performance evaluation has shown the potential of LABIOS' architecture by successfully executing multiple conflicting workloads on a single platform. LABIOS can boost I/O performance on certain workloads by up to 17x and reduce overall execution time by 40-60%. Finally, LABIOS provides a platform where users can express intent with software-defined storage abilities and a policy-based execution. As future work, we plan to further develop our system, test it with larger scales, deploy it on more platforms, and extend its functionality with higher fault tolerance semantics, label dependency graphs, and efficient communication protocols.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grants no. OCI-1835764, CSR-1814872 , CCF-1744317, CNS- 1730488 , CNS-1526887

REFERENCES

[1] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–11.
[2] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. 2010. Energy-efficient cloud computing. *The computer journal* 53, 7 (2010), 1045–1051.
[3] Dimitris Bertsimas and Ramazan Demir. 2002. An approximate DP approach to multidimensional knapsack problems. *Management Science* 48, 4 (2002), 550–565.
[4] Deepavali M Bhagwat, Marc Eshel, Dean Hildebrand, Manoj P Naik, Wayne A Sawdon, Frank B Schmuck, and Renu Tewari. 2018. Global namespace for a hierarchical set of file systems. (July 5 2018). US Patent App. 15/397,632.
[5] Deepavali M Bhagwat, Marc Eshel, Dean Hildebrand, Manoj P Naik, Wayne A Sawdon, Frank B Schmuck, and Renu Tewari. 2018. Rebuilding the namespace in a hierarchical union mounted file system. (July 5 2018). US Patent App. 15/397,601.
[6] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K Lockwood, Vakho Tsulaia, et al. 2016. *Accelerating science with the NERSC burst buffer early user program.* Technical Report. NERSC.
[7] John Biddiscombe, Jerome Soumagne, Guillaume Oger, David Guibert, and Jean-Guillaume Piccinali. 2011. Parallel computational steering and analysis for hpc applications using a paraview interface and the hdf5 dsm virtual file driver. In *Eurographics Symposium on Parallel Graphics and Visualization*. Eurographics Association, 91–100.
[8] M Scot Breitenfeld, Neil Fortner, Jordan Henderson, Jerome Soumagne, Mohamad Chaarawi, Johann Lombardi, and Quincey Koziol. 2017. DAOS for Extreme-scale Systems in Scientific Applications. *arXiv preprint arXiv:1712.00423* (2017).
[9] George H Bryan and J Michael Fritsch. 2002. A benchmark simulation for moist nonhydrostatic numerical models. *Monthly Weather Review* 130, 12 (2002), 2917–2928.
[10] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. 2009. Small-file access in parallel file systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–11.

[11] Chameleon.org. 2018. Chameleon system. https://www.chameleoncloud.org/about/chameleon/. (2018). [Online; accessed 09-14-2018].

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[13] Nathanaël Cheriere, Matthieu Dorier, and Gabriel Antoniu. 2018. *A Lower Bound for the Commission Times in Replication-Based Distributed Storage Systems.* Ph.D. Dissertation. Inria Rennes-Bretagne Atlantique.

[14] Cloud Native Computing Foundation. 2018. NATS Server - C Client. https://github.com/nats-io/cnats. (2018). [Online; accessed 09-14-2018].

[15] Xiaoli Cui, Pingfei Zhu, Xin Yang, Keqiu Li, and Changqing Ji. 2014. Optimized big data K-means clustering using MapReduce. *The Journal of Supercomputing* 70, 3 (2014), 1249–1259.

[16] Matthew L. Curry, H. Lee Ward, and Geoff Danielson. 2015. *Motivation and Design of the Sirocco Storage System Version 1.0.* Technical Report. Sandia National Laboratories. [Online; accessed 09-17-2018].

[17] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. 2016. To Waffinity and Beyond: A Scalable Architecture for Incremental Parallelization of File System Code.. In *OSDI*. 419–434.

[18] Matteo D'Ambrosio, Christian Dannewitz, Holger Karl, and Vinicio Vercellone. 2011. MDHT: a hierarchical name resolution service for information-centric networks. In *Proceedings of the ACM workshop on Information-centric networking*. ACM, 7–12.

[19] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal. 2009. ElasTraS: An Elastic Transactional Data Store in the Cloud. *HotCloud* 9 (2009), 131–142.

[20] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing* 15, 2 (2012), 163–181.

[21] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing*, Vol. 99. 5–33.

[22] Kui Gao, Wei-keng Liao, Arifa Nisar, Alok Choudhary, Robert Ross, and Robert Latham. 2009. Using subfiling to improve programming flexibility and performance of parallel shared-file I/O. In *Parallel Processing, 2009. ICPP'09. International Conference on*. IEEE, 470–477.

[23] Alan Gates. 2012. *HCatalog: An Integration Tool.* Technical Report. Intel®.

[24] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. 2010. Comet: An active distributed key-value store.. In *OSDI*. 323–336.

[25] Joachim Giesen, Eva Schuberth, and Miloš Stojaković. 2009. Approximate sorting. *Fundamenta Informaticae* 90, 1-2 (2009), 67–72.

[26] Google Inc. 2018. CityHash library. https://github.com/google/cityhash. (2018). [Online; accessed 09-14-2018].

[27] Grant, W. Shane and Voorhies, Randolph. 2017. Cereal - A C++11 library for serialization by University of Southern California. http://uscilab.github.io/cereal/. (2017). [Online; accessed 09-14-2018].

[28] Jan Heichler. 2014. *An introduction to BeeGFS.* Technical Report.

[29] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. 2009. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft Research, Redmond, WA.

[30] IBM. 2018. HDFS Transparency. https://ibm.co/2Pciyv7. (2018). [Online; accessed 08-27-2018].

[31] Intel. 2018. Hadoop Adapter for Lustre (HAL). https://github.com/whamcloud/lustre-connector-for-hadoop. (2018). [Online; accessed 08-27-2018].

[32] High Performance Data Division Intel® Enterprise Edition for Lustre* Software. 2014. *WHITE PAPER Big Data Meets High Performance Computing.* Technical Report. Intel. [Online; accessed 08-27-2018].

[33] Kamil Iskra, John W Romein, Kazutomo Yoshii, and Pete Beckman. 2008. ZOID: I/O-forwarding infrastructure for petascale architectures. In *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 153–162.

[34] Laxmikant V Kale and Sanjeev Krishnan. 1996. Charm++: Parallel programming with message-driven objects. *Parallel programming using C+* (1996), 175–213.

[35] Youngjae Kim, Raghul Gunasekaran, Galen M Shipman, David Dillow, Zhe Zhang, and Bradley W Settlemyer. 2010. Workload characterization of a leadership class storage cluster. In *Petascale Data Storage Workshop (PDSW), 2010 5th*. IEEE, 1–5.

[36] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 219–230.

[37] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. IRIS: I/O Redirection via Integrated Storage. In *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS)*. ACM.

[38] Anthony Kougkas, Hariharan Devarajan, Xian-He Sun, and Jay Lofstead. 2018. Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Buffers. In *Proceedings of the 2018 IEEE Cluster Conference (Cluster'18)*. IEEE.

[39] Anthony Kougkas, Hassan Eslami, Xian-He Sun, Rajeev Thakur, and William Gropp. 2017. Rethinking key–value store for parallel I/O optimization. *The International Journal of High Performance Computing Applications* 31, 4 (2017), 335–356.

[40] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–15.

[41] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. 2014. Analysis of federated and global scheduling for parallel real-time tasks. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE, 85–96.

[42] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A high-performance scientific I/O interface. In *Supercomputing, 2003 ACM/IEEE Conference*. ACM/IEEE, Phoenix, AZ, 39–39.

[43] Kenli Li, Xiaoyong Tang, Bharadwaj Veeravalli, and Keqin Li. 2015. Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems. *IEEE Transactions on computers* 64, 1 (2015), 191–204.

[44] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. 2010. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*. ACM, 1–10.

[45] Juan Liu, Yuyi Mao, Jun Zhang, and Khaled B Letaief. 2016. Delay-optimal computation task scheduling for mobile-edge computing systems. In *Information Theory (ISIT), 2016 IEEE International Symposium on*. IEEE, 1451–1455.

[46] Yu-Hang Liu and Xian-He Sun. 2015. LPM: concurrency-driven layered performance matching. In *Parallel Processing (ICPP), 2015 44th International Conference on*. IEEE, 879–888.

[47] Glenn K Lockwood, Damian Hazen, Quincey Koziol, RS Canon, Katie Antypas, Jan Balewski, Nicholas Balthaser, Wahid Bhimji, James Botts, Jeff Broughton, et al. 2017. *Storage 2020: A Vision for the Future of HPC Storage.* Technical Report. NERSC.

[48] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[49] Memached. 2018. Extstore plugin. https://github.com/memcached/memcached/wiki/Extstore. (2018). [Online; accessed 09-14-2018].

[50] Wira D Mulia, Naresh Sehgal, Sohum Sohoni, John M Acken, C Lucas Stanberry, and David J Fritz. 2013. Cloud workload characterization. *IETE Technical Review* 30, 5 (2013), 382–397.

[51] Ron A. Oldfield, Kenneth Moreland, Nathan Fabian, and David Rogers. 2014. Evaluation of Methods to Integrate Analysis into a Large-Scale Shock Physics Code. In *Proceedings of the 28th ACM international Conference on Supercomputing*. 83–92. https://doi.org/10.1145/2597652.2597668

[52] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD Conference on Management of data*. ACM, 1099–1110.

[53] Fengfeng Pan, Yinliang Yue, Jin Xiong, and Daxiang Hao. 2014. I/O characterization of big data workloads in data centers. In *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, 85–97.

[54] Juan Piernas, Jarek Nieplocha, and Evan J Felix. 2007. Evaluation of active storage strategies for the lustre parallel file system. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 28.

[55] Jakob Puchinger, Günther R Raidl, and Ulrich Pferschy. 2010. The multidimensional knapsack problem: Structure and algorithms. *INFORMS Journal on Computing* 22, 2 (2010), 250–265.

[56] Daniel A Reed and Jack Dongarra. 2015. Exascale computing and big data. *Commun. ACM* 58, 7 (2015), 56–68.

[57] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: scaling file system metadata performance with stateless caching and bulk insertion. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, New Orleans, LA, 237–248.

[58] Erik Riedel, Garth Gibson, and Christos Faloutsos. 1998. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*. Citeseer, 62–73.

[59] Robert B Ross, Rajeev Thakur, et al. 2000. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th annual Linux Showcase and Conference*.

[60] Michael W Shapiro. 2017. Method and system for global namespace with consistent hashing. (Oct. 10 2017). US Patent 9,787,773.

[61] Steve Conway. 2015. When Data Needs More Firepower: The HPC, Analytics Convergence. https://bit.ly/2od68r7. (2015). [Online; accessed 08-27-2018].

[62] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*. IEEE, 182–189.

[63] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[64] Devesh Tiwari, Simona Boboila, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines.. In *FAST*. 119–132.

[65] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. 2005. Providing Tunable Consistency for a Parallel File Store.. In *FAST*, Vol. 5. 2–2.

[66] Zhenyu Wang and David Garlan. 2000. *Task-driven computing.* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

[67] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*. Springer, 328–337.

[68] Jean-Francois Weets, Manish Kumar Kakhani, and Anil Kumar. 2015. Limitations and challenges of HDFS and MapReduce. In *Green Computing and Internet of Things (ICGCIoT), 2015 International Conference on*. IEEE, 545–549.

[69] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[70] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories.. In *FAST*. 323–338.

[71] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[72] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[73] Shuanglong Zhang, Helen Catanese, and An-I Andy Wang. 2016. The Composite-file File System: Decoupling the One-to-One Mapping of Files and Metadata for Better Performance.. In *FAST*. 15–22.

[74] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. 2010. PreDatA–preparatory data analytics on peta-scale machines. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.

[75] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: scaling the file system control plane with client-funded metadata servers. In *Proceedings of the 9th Parallel Data Storage Workshop*. IEEE Press, New Orleans, LA, 1–6.

[76] Shujia Zhou, Bruce H Van Aartsen, and Thomas L Clune. 2008. A lightweight scalable I/O utility for optimizing High-End Computing applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, Miami, FL, USA, 1–7.