A Software-Defined Storage for Workflow Applications

Samer Al-Kiswany
University of Waterloo

Matei Ripeanu
University of British Columbia

Abstract. We present a software-defined storage architecture with two key properties: firstly, it enables external control of storage operations, and, secondly, it allows extending the storage system with new, workload-specific, optimizations, all without breaking the file system abstractions. We argue that this architecture is generic, we prototype FlexStore following this architecture, we instantiate it to support workflow applications, and we report on our preliminary experience.

1 Introduction

Software defined networks (SDN) enable external control of network switch operations while providing an incremental adoption path by continuing to support traditional network operations. This approach facilitated investigating new research frontiers and optimizing the network operations for specific workloads while maintaining backward compatibility.

Similarly to software defined networks we propose *a software-defined storage architecture* that enables experimentation and workload specific optimization while providing, as well, an incremental adoption path. The architecture we propose offers two key properties: firstly, it enables *external control of storage system operations*, and, secondly, it allows *extending the storage system with new, workload-specific optimizations*, all without breaking the file system abstractions.

The architecture we propose divides the storage system into three planes: the *primitives*, *operations*, and *control planes*. The *primitives plane* provides the primitive storage system operations such as object level operations, data transfer, metadata operations, system status retrieval, and caching. The *operations plane* implements, using these primitives, complete storage system operations (e.g., the write data path, space allocation). The operations plane can include multiple implementations of the same operation (e.g., different space allocation policies, or different replication algorithms). The *control plane* allows application control over which version of an operation to invoke per-file. Finally, to address the wide range of usage scenarios, the operations plane is extensible, allowing adding new implementations of operations.

We prototyped the ***Fl****exible and* ***Ex****tensible* ***Stor****age System: FlexStore*, based on the proposed architecture and extended it with storage operations optimized for workflow applications (described in detail in §2). While we believe that *external control* and *extensibility* are the key attributes that make this storage architecture appealing, we have also evaluated the performance gains our prototype brings to demonstrate its ability to efficiently support known optimizations. Compared to two widely used distributed storage systems Ceph [1] and GlusterFS [2], *FlexStore* offers, sizeable gains: up to 6x higher performance and 10x lower network overhead, at the same time, for real applications *FlexStore* offers 1.25 - 1.7x better performance.

2 Background and Motivation

This section starts by briefly setting up the context (§2.1), presents a summary of workflows data access patterns (§2.2). and presents other scenarios where the customization of the storage system supported by *FlexStore* is essential (§2.3).

2.1 Intermediate Storage and Workflow Applications

Workflow applications assemble a set of standalone executables to perform complex data processing [3-6]. The executables communicate through files stored on a shared file system. Files generated during the workflow execution are temporary (i.e., they are not needed after workflow completion). Consequently, they are often stored in a 'scratch' shared storage dedicated to the application. To avoid accessing the off-cluster backend storage system this scratch space is often obtained by aggregating the storage resources of the compute nodes allocated to the application, a technique called *intermediate storage* (Figure 1). This deployment scenario has become popular [7].

The scheduler schedules the individual tasks when all their input files are available. Through the DAG the scheduler can infer the files' access patterns before workflow startup.

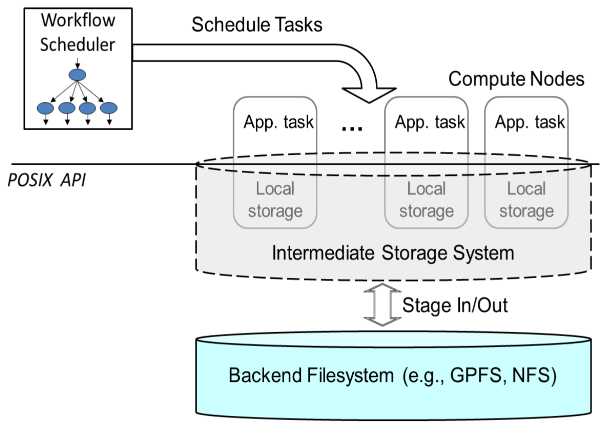

Figure 1. **Target deployment scenario.** The scheduler executes tasks on compute nodes that access files at intermediate storage. Input/output data is staged in/out from the backend storage.

Although workflow applications generate well studied data access patterns [3-6] it is difficult to optimize the intermediate storage for the generated I/O workload as different files in the same workflow have different access patterns (summarized in §2.2). Consequently, system-wide configurations/optimizations are not useful. Thus, the requirement to support *per-file* optimizations and handing explicit control over data placement, caching, replication, and space allocation policies to the workflow scheduler. These optimizations can bring sizable gains as shown by previous studies [8][19] and by our own evaluation [13].

In addition to supporting workflow applications, an intermediate storage is often used in other deployment scenarios that would all benefit from application-specific optimizations, section (§2.3) discusses few examples.

2.2 Workflows Common Data Access Patterns

We detail a few common data usage patterns identified by past studies [3-6], and the opportunities for optimizations:

- *Pipeline*: an optimized space allocation stores the pipeline files on the local storage node on the same machine of the tasks to increase access locality.
- *Broadcast*: A single file is used by multiple tasks. An optimized replication mechanism can replicate the shared file to eliminate the possibility of hot spots.
- *Reduce*: An optimized space allocation operation can place all reduce files on one node.

2.3 The Need for Extensibility

In addition to intermediate storage for workflow applications, numerous other scenarios can benefit from control over storage system operations. As it is challenging (if not impossible) to build a single storage system that satisfies all these scenarios, the software-defined storage architecture can allow extending the storage system with future operations. Two examples follow:

- *Checkpointing*. Checkpointing is an indispensable fault tolerance technique, which generates an intense I/O workload. A number of optimizations (e.g., deduplication) have been proposed [9] to reduce I/O pressure, reduce checkpoint size, or to simplify the management of a checkpointing repository, but all require identifying the checkpointing files and the checkpointing policy. A software-defined storage can support checkpointing workloads by: *(i)* optimizing the storage operations (e.g., a data path with deduplication), or a file close() operation that only maintains the last *N* versions of the checkpoint files, and *(ii)* handing control over these operations to the platform.
- *Custom deployment scenarios*. Previous work proposes dedicating a set of storage nodes for format transformation [10], as a stage-out buffer [11], or as a data-center scratch space [12]. Higher efficiency can be achieved in these deployment scenarios via a software-defined storage system that enables application specific data placement, caching, or retention policy.

3 A SOFTWARE-DEFINED STORAGE ARCHITECTURE

This section presents the insights (§3.1) that led to the proposed architecture (§3.2) and to *FlexStore* design (§3.3).

3.1 Insights

In addition to the specific workload characteristics (§1.1) and deployment practices (§2.1), the proposed architecture is made feasible by the following two key observations:

- First, *ability to isolate operation implementation*: performance-critical storage operations (e.g., space allocation, replication, the data path, caching) can be well modularized with no interaction between operations (with the exception of updating the file metadata). This makes it possible to create multiple implementations of storage operations and select one of them at runtime.
- Second, *a common 'primitives' substrate*: the storage system relies on well-defined common primitives that encapsulate data and metadata operations. We encapsulate these in the primitives plane (described next).

3.2 System Architecture Overview

The storage system has three main components: a centralized metadata manager that maintains the metadata and system's soft state, the storage nodes that store the data, and the client's file system (CFS) which provides the client-side POSIX file system interface.

The architecture (Figure 2) presents the storage system as three distributed planes (*primitives*, *operations*, and *control* plane). Each plane functionality is distributed across the three components (manager node, storage nodes, and CFS).

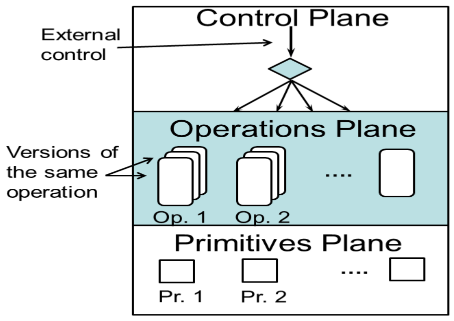

Figure 2. System Architecture. The *operations plane* (in gray) is the extensible plane. The control *plane enables* aplication control.

The primitives plane provides the basic primitives for building a storage system, these are, in principle, low level abstraction for the lower hardware/software operations, for instance: storage abstraction with simple object put/get/delete API.

The operations plane uses the functionality offered by the primitives plane to implement complete storage operations, for instance: space allocation at the manager, or replication at storage nodes. The operations plane may include multiple implementations of the same operation. These implementations have the same application-level semantics but differ internally (e.g., in terms of space allocation policy, or replication protocol), thus, offering different performance or reliability properties. The operations plane can be extended with new versions of the operations.

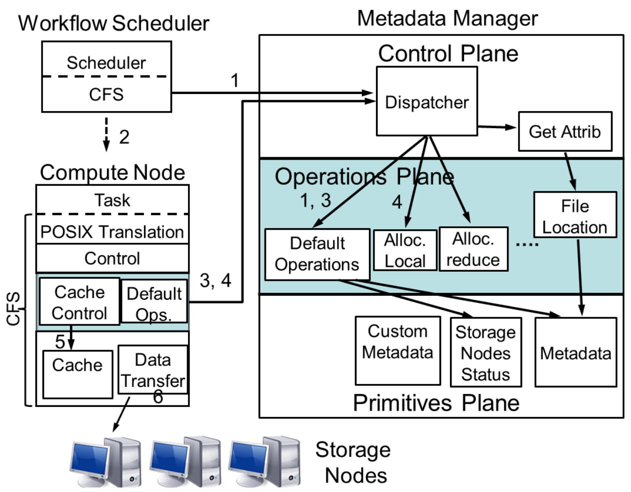

Figure 3. FlexStore design. The figure: (1) presents the design of the FlexStore metadata manager (full detail, right box, note that the CFS and storage nodes, follow the same layered design); and (2) presents FlexStore deployment as an intermediate storage system over the allocated compute nodes (details in §3.5). The workflow scheduler and the workflow tasks access the intermediate file system through the CFS (low left). Lines represent internal invocations (possibly on remote nodes).

The control plane provides the ability for the scheduler, to control which operation implementation is invoked and to pass control parameters to specific operations. The operations plane has a default non-optimized implementation of every operation which is used if no control information is provided.

3.3 FlexStore Design

Figure 3 shows the *FlexStore* design. The rest of this section details the design of the system three components.

CFS Design. CFS provides a subset of the POSIX file system API: mainly the file/directory create/read/write/ delete/getattrib API. The CFS translates the application POSIX calls to *FlexStore* operations. The file metadata is committed to the manager only on close() system call.

The CFS primitives plane provides caching (with pin/unpin interface) and file transfer to storage nodes primitives; while the operations plane implements default object-get/put operations, and default metadata get/put operation, and an cache control optimized for the pipeline pattern.

Storage Node Design. Storage nodes implement a simple object storage system with simple get/put/delete functionality. On object-put the node can replicate the object to other nodes and implement a range of consistency semantics depending on the file custom attributes.

The storage nodes' primitives plane provides primitives for local-disk put/get/delete, and object transfer to another node. The operations plane includes a default implementation for object-get/delete/put operations (i.e., unreplicated put), and two optimized versions of the object-put operations: put with eager parallel replication used in broadcast pattern and put with asynchronous chain replication used for reliability).

Manager Node Design. The manager maintains the files metadata, and the storage nodes soft state (per node free space). The manager primitives plane provides primitives for metadata update and retrieval, storage nodes' status retrieval, and persistent key-value API for storing operations' custom data. The operations plane includes five implementations of the space allocation operation (each optimized for a different workflow access pattern).

The control plane in all components selects which operation version to use for a given request based on the file extended attributes. Further, these attributes are piggybacked on all communications between the system components, enabling the implementation of distributed optimized operations.

Extending the Storage System. To extend the system with a new version of an operation, the developer needs to choose a tag (extended attribute) that will trigger the optimized operation, and implement the callback function on all system components related to the optimization. While it is hard to quantitatively measure extensibility, we have verified that it is possible to implement all optimizations proposed in §2.3.

Finally, the default storage system behavior if no custom attributes are used is an unreplicated storage system with round robin allocation policy, and LRU caching.

3.4 Runtime Flexibility: Controlled Operations

Design for Runtime Flexibility. The control plane for all storage system components follow a '*dispatcher'* design pattern: based on the requested operation and the file extended attributes, the dispatcher dispatches the request to the specific version of an operation. Figure 3 shows the detailed internal design of the metadata node (the client CFS and storage node follow the same design).

Control Interface. To ensure cross-compatibility we overload Linux extended file attributes API: setx-/getxattr().

Cross compatibility. We note that the proposed design is cross compatible. A traditional workflow scheduler that does not use extended attributes to control the storage operations can still use the default system implementation. Similarly, a modified scheduler can still work on traditional storage system, although the used extended attributes will not change the storage system behavior. We argue that this solution unlocks an incremental adoption path in which schedulers and storage systems can incrementally add support for software controlled operations.

3.5 Putting it all Together

Figure 3 illustrates how the integrated system operates. The workflow scheduler indicates (through adding custom attributes) the usage pattern of every file before the task producing that file is scheduled (path labeled 1 in Figure 3). Next the scheduler schedules the task on a compute node (2). When the task opens the file for write, the CFS retrieves the file custom attributes (3) and requests a space allocation from the manager (4) using those attributes. The manager uses the attributes to guide the data placement. The manager sends to the CFS a set of storage nodes to write to. The CFS starts the data write operation. The CFS uses the custom attributes to select the desired version of the write operation at the client. Further, the CFS passes the attributes to the storage nodes to select storage node write operation implementations (for instance selecting or parametrizing the replication mechanism). Once the file is created, its location is exposed through the ("location") extended attribute that has value for every file in the system (thus enabling location-aware scheduling).

4 EVALUATION

We refer the reader to our technical report for detailed evaluation with syntactic and real applications [13].

Testbed. We used 101 nodes (404 cores) on Grid5000 'Nancy' site. Each node: Intel Xeon X3440 4 core, 2.5-GHz CPU, 16GBRAM, 1Gbps NIC, and 320GB SATA II disks.

Synthetic Benchmarks. We designed a set of synthetic benchmarks to mimic the access patterns described in §2.2. Overall, *FlexStore*, exhibits close to optimal performance (estimated by using hardcoded I/O paths different for each benchmark) and offers higher performance than FlexStore-D (the default *FlexStore* implementation without any optimizations), which, in turn, has better performance than NFS, Ceph and GlusterFS. This shows that the overheads brought by the architecture are paid off by the performance improvements. *FlexStore* brings 6x higher performance (10x lower network overhead) for pipeline, 2x better performance and 87% lower network load for reduce, and 2x better performance and 7% lower network load for broadcast) compared to FlexStore-D.

Real Applications. We evaluated our prototype using three real applications: Montage, BLAST, and ModFTDoc [13]. These applications use a mix of the aforementioned access patterns for different files. *FlexStore* efficiently used different optimization for different files in the workflows and brought 1.25 - 1.7x better performance for the three application. Our analysis shows that the current prototype adds 7% scheduling overhead caused by implementation shortcuts that can be easily overcome in the next version.

5 RELATED WORK

Previous efforts in building software-defined storage focused on separating the control and data planes to enable application-informed control of storage operations [14-17, 20]. They are either designed with specific optimization in mind (e.g., IOFlow focuses solely on providing quality of service) or provide a one-size-fit-all treatment for all application files. Unlike previous effort, *FlexStore* enables per-file control of all storage operations, and enables extending the storage with new versions of storage operations. BAD-FS file system [18] is the closest in principle to our system. BAD-FS allows the scheduler to configure the caching, consistency and replication policies per volume to better serve pipeline based workflows. Unlike our system, BAD-FS supports only pipeline access, is not extensible, and controls storage policies per volume not per file.

6 REFERENCES

1. Sage Weil, et al. *Ceph: A Scalable, High-Performance Distributed File System*. *OSDI* 2006.
2. *GlusterFS*. cited 2016; https://www.gluster.org/.
3. Wozniak, J. and M. Wilde. *Case studies in storage access by loosely coupled petascale applications*. *Petascale Data Storage Workshop*. '09.
4. Shibata, T., et al. *File-access patterns of data-intensive workflow applications and their implications to distributed filesystems*,*HPDC* '10.
5. Bharathi, S., et al., *Characterization of Scientific Workflows*, in *Workshop on Workflows in Support of Large-Scale Science*. 2008.
6. Yildiz, U., , et al.,*Towards scientific workflow patterns*, in *Workshop on Workflows in Support of Large-Scale Science*. 2009.
7. *Trinity/NERSC-8 Use-Case Scenarios*. 2013 [cited 2014.
8. Vairavanathan, E., et al. *A Workflow-Aware Storage System: An Opportunity Study*. *CCGrid* 2012.
9. Al-Kiswany, S., et al. *stdchk: A Checkpoint Storage System for Desktop Grid Computing*. in *Conference on Distributed Computing Sys*. 2008.
10. Min, L., et al., *Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures*, *Supercomputing*. 2010.
11. Ramya, P., et al., *Provisioning a Multi-tiered Data Staging Area for Extreme-Scale Machines*, *International Conference on Distributed Computing Systems, 2011*.
12. Henry, M.M., R.B. Ali, and S.V. Sudharshan, */scratch as a cache: rethinking HPC center scratch storage*, *Supercomputing*. 2009.
13. Al-Kiswany, S., et al. *The Case for Cross-Layer Optimizations in Storage: A Workflow-Optimized Storage System*. *Technical Report. The University of British Columbia. 2016*.
14. Eno, T., et al., *IOFlow: a software-defined storage architecture*, SOSP 2013.
15. Darabseh, A., et al. *SDStorage: A Software Defined Storage Experimental Framework*. in *International Conference on Cloud Engineering*. 2015.
16. Gracia-Tinedo, R., et al., *IOStack: Software-Defined Object Storage.* IEEE Internet Computing, 2016.
17. Alba, A., et al., *Efficient and agile storage management in software defined environments.* IBM Journal of Research and Dev. 2014.
18. Bent, J., et al. *Explicit Control in a Batch-Aware Distributed File System*. *NSDI* 2004.
19. Costa, L., et al., *The Case for Workflow-Aware Storage: An Opportunity Study*, Journal of Grid Computing, 13(1): 95-113 (2015)
20. Al-Kiswany, S., et al., *The Case for a Versatile Storage System*, HotStorage 2009.