A Workflow-Aware Storage System: An Opportunity Study

Emalayan Vairavanathan[*], Samer Al-Kiswany[*], Lauro Beltrão Costa[*], Zhao Zhang[++], Daniel S. Katz[+], Michael Wilde[+], Matei Ripeanu[*]

[*] The University of British Columbia
{emalayan, samera, lauroc, matei}@ece.ubc.ca

[+] University of Chicago & Argonne National Laboratory
{dsk, wilde@ci.uchicago.edu}

[++]University of Chicago {zhaozhang@uchicago.edu}

***Abstract* — This paper evaluates the potential gains a *workflow-aware storage system* can bring. Two observations make us believe such storage system is crucial to efficiently support workflow-based applications: First, workflows generate irregular and application-dependent data access patterns. These patterns render existing storage systems unable to harness all optimization opportunities as this often requires conflicting optimization options or even conflicting design decision at the level of the storage system. Second, when scheduling, workflow runtime engines make suboptimal decisions as they lack detailed data location information.**

This paper discusses the feasibility, and evaluates the potential performance benefits brought by, building a workflow-aware storage system that supports per-file access optimizations and exposes data location. To this end, this paper presents approaches to determine the application-specific data access patterns, and evaluates experimentally the performance gains of a workflow-aware storage approach. Our evaluation using synthetic benchmarks shows that a workflow-aware storage system can bring significant performance gains: up to 7x performance gain compared to the distributed storage system - MosaStore and up to 16x compared to a central, well provisioned, NFS server.

Keywords – Large-scale storage systems, workflow-aware storage systems, workflow optimizations.

I. INTRODUCTION

Meta-applications that assemble complex processing workflows using existing applications as their building blocks are increasingly popular. Examples include various scientific workflow applications (e.g., modFTDock[1], Montage [2], PTMap[3]) and even map-reduce applications [4].

While there are multiple ways to support the execution of these workflows on large clusters, in the science area—where a large legacy codebase exists—one approach has gained widespread popularity: a *many-task approach* [5] in which application workflows are assembled from independent, standalone executables that communicate through intermediary files stored in a shared datastore or independently, on the local storage of the nodes where task execution takes place.

The main advantages of this approach, adopted by most existing workflow runtime engines (*e.g.*, Swift [6], Pegasus [7]) are simplicity, direct support for legacy applications, and natural support for fault tolerance. First, a shared file system approach simplifies workflow development, deployment and debugging: workflows can be developed on a workstation then deployed on a cluster without changing the environment. Moreover, a shared storage system simplifies workflow debugging as intermediate computation state can be easily inspected at runtime and, if needed, collected for performance profiling. Second, most of the legacy applications that form the individual workflow stages are coded to read their input through the POSIX API. Finally, compared to approaches based on message passing, communicating between workflow stages through a storage system that offers persistency makes support for fault-tolerance much simpler: a failed execution step can simply be restarted on a different compute node as long as all its input data is available in the shared storage.

Although these are important advantages, the main drawback of this approach is decreased performance: the abstraction of a shared file-system (reached through a POSIX API) constrains the ability to harness various performance-oriented optimizations and to communicate between the storage layer and the workflow runtime engine. Specifically, a traditional file system cannot use the information available at the level of the workflow execution engine to guide the per-object data placement or to optimize for various access patterns at per-object granularity. Similarly, a traditional distributed storage system does not expose data-location information, which prevents the workflow runtime engine from exploiting opportunities for collocating data and computation.

This paper investigates the feasibility and the performance benefits of a *workflow-aware storage system*: a storage system that aggregates the resources of the computing nodes (*e.g.*, disks, SSDs, and

memory) and has two key features. First, it is able to efficiently support the data access patterns generated by workflows through optimizations at the file or directory level. Second, it exposes data placement information so that the workflow runtime engine can make data-aware scheduling decisions.

To support specific data access patterns, the storage system will use hints [8] that will drive the data layout (e.g., co-placement, replication levels, chunk placement) that indicate the expected access pattern. In §II.B we argue that such hints can be either provided directly by the workflow runtime engine, as it has full information about the data usage patterns; or inferred by the storage system itself based on historical information.

The goal of this paper. We believe that a *workflow-aware storage system* that supports specialized, per-file optimizations is essential to efficiently supporting workflow applications as their generated data access patterns are irregular and application dependent.

Our goal with this opportunity study is to *evaluate the performance benefits of a workflow-aware storage system before paying the full cost of prototyping it*. To this end, we have started from the MosaStore storage system (§III.A) and added the ability to expose data location through POSIX's extended file attributes. This enables MosaStore integration with a workflow runtime engine that supports data-aware scheduling. Additionally, to understand the performance impact of per-file optimizations we use the following methodology: we start from the workflow access pattern characterization provided by Wozniak *et al.*[9] and derive the file-level optimizations the storage system needs to support (§II.A). Once these are identified, we change MosaStore to support them (§III.B). We stress however, that our changes at this point do not amount to a new system design – they are generally hardcoded paths in the storage with the single purpose of supporting our evaluation experiments (§III.C and §III.D).

Our results. Our exploration shows that building a workflow aware storage system is indeed feasible and can bring significant performance gains. It is feasible, as previous studies showed that the workflows have a small set of common data access patterns, thus a small set of storage optimizations are enough to serve these patterns and our preliminary study shows that these optimizations can be incorporated in a high performance storage without significantly increasing the complexity of the system. Additionally, our evaluation using synthetic benchmarks shows that a workflow-aware storage system can bring significant performance gains. Compared to a general distributed system that uses the same hardware resources, per-file optimizations and exposing data location enable 1.3x to 7x performance gains depending on the access pattern. Further, compared to a central NFS server deployed on a well provisioned server-class machine (with multiple disks, and large memory), a workflow-aware storage system achieves up to 16x performance gains. (NFS only provided competitive performance under cache friendly workloads due to it well provisioned hardware.)

Finally, we note that a number of alternative approaches (§II.C) have been proposed to alleviate the storage bottleneck for workflow applications. They range from storage glide-ins (*e.g.*, BADFS [10]) to building application-optimized storage systems (e.g. HDFS [11], BADFS [10]), to building a configurable storage system that is tuned at deployment time to better support a specific application [12], to offering specific data access optimizations (*e.g.*, location-aware scheduling [13], caching, and data placement techniques [14]). Taken in isolation, these efforts do not fully address the problem we face as they are either specific to a class of applications (*e.g.*, HDFS for map-reduce applications), and consequently incapable to support a large set of workflow applications; or enable system-wide optimizations throughout the application runtime, thus inefficiently supporting applications that have different usage patterns for different files. Our solution integrates lessons of the above past work and provides mechanisms to support storage optimizations at the file/directory granularity.

II. BACKGROUND AND RELATED WORK

A. Data Access Patterns in Workflow Applications

Several studies explore the data access patterns of workflow applications: data access patterns of large group of scientific workflows are studied and characterized by Wozniak *et al.* [9] (*5 applications*), Daniel S. Katz *et al.* [15] (*12 applications*), Shibata et al. [16] (*5 applications*), Bharathi, *et al.* [17] (*5 applications*) and Ustun *et al.* [18].This subsection briefly presents the common data access patterns identified by these studies and the opportunities for storage optimizations at the data-object level they generate (summarized in Table 1):

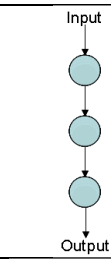
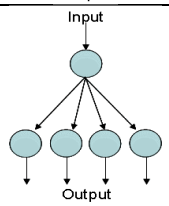
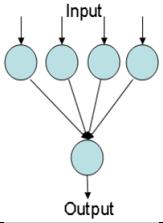
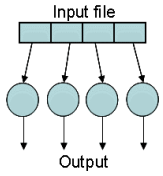
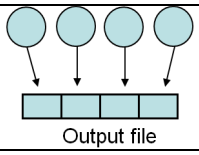
- *Pipeline*: A set of compute tasks are chained in a sequence such that the output of one task is the input of the next task in the chain. An optimized storage system can store the intermediate files on a storage node on the same machine as the one that executes the task (if space is available) to increase access locality and efficiently utilize local caches. Ideally, the location of the cached copy is exposed to the workflow scheduler so that the task that consumes this data is scheduled on the same node.
- *Broadcast*: A single file is processed by a number of compute nodes at the same time. An optimized storage system can create enough replicas of the shared file to eliminate the possibility that the node(s) storing the file become overloaded – resulting in a performance bottleneck.
- *Reduce*: A single compute task has as its input files that are produced by multiple different computations. Examples include a task that checks the results of previous tasks for a convergence criterion, or a task that calculates summary statistics from the output of many tasks. An optimized storage system can intelligently place all these input files on one node and expose their location information, thus creating an opportunity for scheduling the reduce task on that node and increasing data access locality.
- *Scatter/Gather*: A single file is written to/read from by multiple compute nodes where each node accesses a disjoint region in the file. An optimized object storage system can optimize its operation through setting the storage system block size to the application file region size, and through intelligent placement of file's blocks, and by exposing block-level data placement.

B. Determining the Data Access Patterns

Information on the data access patterns is crucial to enable the ability of the storage system to optimize. Most of the applications use thousands of files and contain more than one pattern. Several approaches already described in past work can be used to provide such information to a workflow-aware storage system: workflow runtime engine can provide this information itself as this information may be available in the workflow description, this information can be inferred by the storage system itself through monitoring, or inferred by profiling the application's IO operations. This section describes briefly these approaches:

- *Application analysis by the workflow runtime engine.* Workflow runtime engine builds and maintains the data dependency graph and uses this graph to schedule the computation once the data become available. Thus, the runtime engine already knows the usage patterns and the lifetime of every file in the workflow execution. This information can

Table 1. Popular data access patterns generated by workflows. Circles represent computations. An outgoing arrow indicated that data is produced (through a temporary file) while an incoming arrow indicates that data is consumed (from a temporary file). There may be multiple inputs and outputs via multiple files. The notation we use is similar to that used by Wozniak *et al.* [9].

Pattern	Pattern Details	Optimizations
Pipeline		▪ Node-local data placement (if possible). ▪ Caching. ▪ Data-informed workflow scheduling.
Broadcast		▪ Optimized replication taking into account the data size, the fan-out, and the topology of the interconnect.
Reduce		▪ Reduce-aware data placement: co-placement of all output files on a single node; ▪ Data-informed application scheduling
Scatter		▪ Application-informed block size for the file. ▪ Application-aware block placement; ▪ Data-informed application scheduling
Gather		▪ Application-informed block size for the file. ▪ Application-aware block placement;

be provided to the underling storage system to optimize its operations based on these hints. Santos-Neto et al. [8] propose using 'tags' as the cross-layer communication mechanism through which the workflow runtime engine can provide hints to the storage system on data access patterns. These hints can be communicated as extended file attributes to comply with the POSIX API. In fact, file attributes can be used as a bidirectional communication channel. On one side the workflow engine can pass data usage hints to the storage system, on the other side storage system can expose per file/directory internal information (e.g., data location) that help workflow engine optimize its runtime decisions.

- *Monitoring and auto-tuning*. Although the dependency graph provides the usage patterns present in an application that can inform which optimization should be used (*e.g.,* broadcast pattern indicates the need of replication), the storage system may need more information to fine tuning the optimization (*e.g.,* how many replicas is the optimal). Past work [19] uses a monitoring module to collect information on the access patterns and predicts the future ones. The storage system can optimize its operations based on these predictions. Note that the auto-tuning is out of the scope of this paper and is an ongoing work of our group [20].
- *Application profiling*. Another approach to fine tuning the configuration is having the system administrator inferring the access patterns through application profiling or from a description provided by the application developer. The administrator then configures the storage system to optimize its operation for the frequently used access patterns [12].

C. Related Work

The volume of past work on alleviating storage system bottlenecks is humbling. This section limits itself to positioning a smaller number of projects that directly focus on alleviating the storage bottleneck for workflow applications. The first two directions we discuss below optimize for performance while preserving the shared storage system abstraction. The third category of related work co-designs the workflow runtime engine and the storage system – this approach holds the promise of higher performance yet it leads to a more complex design by breaking the layering between the storage and the workflow runtime which offers a natural abstraction and separation of concerns.

Application-optimized storage systems. Building storage systems geared for a particular class of I/O operations or for a specific access pattern is not uncommon. For example, the Google file system [21] optimizes for large datasets and append access, HDFS [11] which shares many design goals with GPFS-SNC [22], optimizes for immutable data sets, location-aware scheduling and rack-aware fault tolerance; the log-structured file system [23] optimizes for write intensive workloads, arguing that most reads are served by ever increasing memory caches; finally BAD-FS [10] optimizes for batch job submission patterns. These storage systems and the many others that take a similar approach are optimized for one specific access pattern and consequently are inefficient when different data objects have different patterns, like in the case of workflows.

Highly Configurable Storage Systems. A few storage systems are designed to be highly configurable – and thus, after deployment-time (re)configuration, efficiently serve a wide set of applications. The versatile storage system [12] argues that storage systems should be specialized for the target application at deployment time. In the same vein, Ursa Minor [24] offers deployment-time configurability to meet application access patterns and reliability requirements. While these storage systems can be configured to better support a range of applications, they are not designed to support workflows with different access patterns for different files.

Co-designing the workflow runtime engine and the data-store. A number of projects (e.g., Falkon[25, 26], AME [27]) give up the shared storage system abstraction as well as the POSIX interface and redesign, from scratch, a minimal storage system coupled with the workflow runtime engine. The storage system implemented gives up the shared namespace offered by a POSIX-compliant shared storage system and treats each participating node as an independent, fully functional storage element. An independent service keeps track of data location. The scheduler will use this service and attempt to submit jobs where data is already located or orchestrate explicit data moves so that data is available on the local storage of a node when a task starts executing. While this approach is likely to lead to higher performance (an observation that holds for designs that give up layering) we believe that its drawbacks are non-negligible (higher system complexity and limited or failure to support large files that do not fit in the local storage of a single node), in addition to forfeiting most of the advantages of a layered design which we summarize in the introduction. .

III. An Evaluation of Opportunities

To evaluate the feasibility and performance of a workflow-aware storage system we customized MosaStore storage system [12] to enable evaluating optimizations for the access patterns presented in §II.B. The rest of this section presents briefly MosaStore design (§III.A) such that this paper is self-contained, describes the various changes we make to

the system (§III.B) to support per file optimizations, and presents our opportunity study (§III.C and §III.D).

A. MosaStore Design

MosaStore (http://mosastore.net) is a distributed storage system that can be configured to match specific workload characteristics. MosaStore adopts an object-based distributed storage system architecture, with three main components: a centralized metadata manager, the storage nodes, and the client's system access interface (SAI) which uses FUSE [28] kernel module to provide a POSIX file system interface.

Each file is divided into fixed size blocks that are stored on the storage nodes. The metadata manager maintains a block-map for each file which contains the file's blocks information including the hash value of every block. The SAI implements the client-side POSIX interface for accessing the system.

Data placement. MosaStore uses a round robin blocks placement policy. When a new file is created on a stripe of *n* nodes the file's blocks are placed in a round robin fashion across these *n* nodes.

Replication. MosaStore uses lazy replication for data reliability. When a new data block is produced by the application, the MosaStore client stores it at only one of the storage nodes and instructs storage nodes to lazily replicate the new block to other storage nodes.

B. Hacks: Customizing MosaStore

A workflow-aware storage system should provide per-file configuration at run time to support high-configurability for diverse applications access patterns. Further, the workflow-aware storage system should be workflow engine friendly. That is, it should expose internal per-file/directory information (e.g. data location) that help the workflow engine optimize its runtime decisions (*e.g.*, data location aware scheduling).

To mimic a workflow-aware storage system and to evaluate its performance, we customized MosaStore for each pattern we evaluate. This section briefly presents these hardcoded customizations (described in more detail in §III.C in the context of the evaluation experiments).

Our goal with these experiments is to better understand the potential performance gains that can be offered by a workflow-aware storage system before completely implementing one. Thus, for some of the customizations we make that are incompatible with each other in the current MosaStore implementation (*e.g.*, different data placements schemes as they are, in the original MosaStore, system-wide policies rather than per-file policies) we enable/disable for each experiment some of these changes in the code, recompile the code, and redeploy the storage system.

For all optimizations we describe below we harness the fact that MosaStore exposes data placement through POSIX extended file attributes and we assume that the workflow runtime engine can optimize its decisions using this information (*i.e.*, the runtime engine can schedule computations close to where data is located).

- *Optimized data placement for the pipeline pattern*. We changed the MosaStore data placement module to prioritize storing output files produced by a pipeline stage at the same node where the task corresponding to that stage runs. If data does not fit on the local node, file blocks are shipped remotely through the normal MosaStore mechanisms.
- *Optimized data placement for the reduce pattern*. We changed MosaStore to co-locate all the output files of a workflow stage followed by a reduce stage on a single pre-specified storage node. If data does not fit on the local node, file blocks are shipped remotely through the normal MosaStore mechanisms.
- *Replication mechanism optimized for the broadcast pattern*. To avoid that the storage nodes for a file used in a broadcast pattern become a bottleneck, we increase the replication factor of these files. We changed the MosaStore lazy replication mechanism to eager parallel replication: replicas are created eagerly while each block is written to storage.
- *Data block placement for the scatter and gather patterns*. Unlike other patterns described above that require optimizations at file level, scatter and gather require block-level optimizations, as a single file's blocks are accessed by a number of compute tasks in parallel. Consequently, we set the MosaStore block size to match the application per-node access region size, and constrain the MosaStore data placement such that we can determine where each block of a file is placed. Further, we hardcode the scheduling decision to run the compute task on the node that has the specific file block accessed by that task.

In addition to the per-pattern customizations described below, we have applied one general optimization: we prioritize local file access to increase access locality. We changed the storage client to prioritize reading blocks (if available) directly from the disk of a local node instead of reading from remote storage nodes.

C. Experimental Setup

We ran our evaluation on a cluster of 20 machines. Each machine has Intel Xeon E5345 4-core, 2.33-GHz CPU, 4-GB RAM, 1-Gbps NIC, and a 300-GB 7200-rpm SATA disks. The system has an additional NFS server that runs on a well provisioned machine with an Intel Xeon E5345 8-core, 2.33-GHz CPU, 8-GB RAM, 1-Gbps NIC, and a 6 SATA disks in a RAID 5 configuration.

Throughout our evaluation we compare the performance of the following intermediate storage alternatives: a workflow-aware storage system (i.e., the data access pattern optimized MosaStore); a generic distributed storage system (we use an un-optimized MosaStore deployment); and an NFS server representing a central shared storage system that often is found in large scale computing machines. We note that an un-optimized MosaStore storage system is similar in architecture and design to a set of cluster storage systems such as Luster. Further, although NFS is not typically used in large scale platforms, in our setup of 20 machines, it models, at our scale, a well provisioned shared central storage systems like the ones available in found in large scale platforms.

The cluster is used to run one of the shared storage systems (MosaStore with either the default code or with the changes we have made to mimic a workflow-aware storage system) and the synthetic applications. One node runs the MosaStore manager and 19 run the storage nodes, the MosaStore SAI, and the application itself. With the NFS configuration we run NFS on the above mentioned server and the application on the other 19 nodes. We run the experiments at least 10 times and report averages with 95% confidence and a 5% error margin..

Current workflow systems often work as follows: they stage-in the input data from a backend storage system to an intermediate shared storage space, process the data in this shared space, and then stage-out the results and persist them on the back-end system. Our set of synthetic application benchmarks fit this pattern and are composed of read/write operation that mimic the file access patterns described earlier. Figure 1 summarizes these benchmarks.

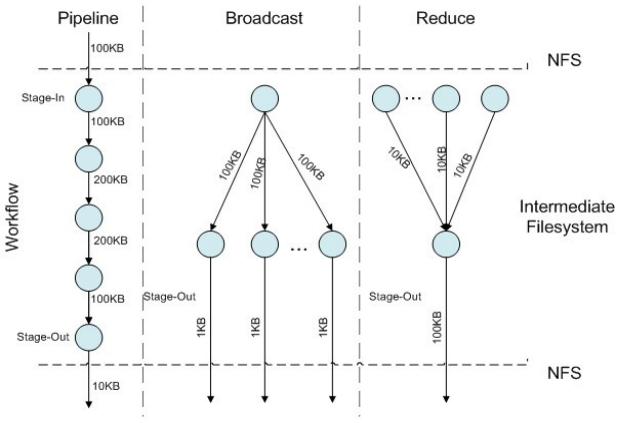


Figure 1.Summary of synthetic benchmarks for pipeline, broadcast and reduce patterns. Nodes represent workflow stages (or stage-in/out operations) and arrows represent data transfers through files. Labels on the arrows represent file sizes for the '*small'* workload. The other workload sizes are presented in Table 2.

Table 2 - File sizes for different workflow patterns.

Workload	**Pipeline (in, intermediate & out)**	**Broadcast (in & out)**	**Reduce (in & out)**
Small	100KB, 200KB, 10KB	100KB, 1KB	10KB, 200KB
Medium	100 MB, 200 MB, 1MB	100 MB, 1MB	10MB 200 MB
Large	1GB, 2GB, 10MB	1 GB, 10 MB	100MB, 2 GB

D. Experiments and Results

This section presents the experiments we executed, the MosaStore customizations that support them, and the performance results.

1) Pipeline Pattern Evaluation

Customization. To efficiently support the pipeline pattern the workflow-aware storage system changes the MosaStore data placement mechanism to place newly created files on the node that produces them. This change efficiently supports fast access to the temporary files used in the pipeline pattern as the next stage of the pipeline is launched on the same node.

The workload (Figure 1 left). We run in parallel 19 application pipelines similar to the ones described in the figure. Each of these pipelines stages-in a common input file from the shared back-end storage (i.e., the NFS server), goes through three processing stages, that read input from the intermediate store and write the output to the intermediate store, then the final output is staged out back to back-end (i.e., NFS).

The cluster is used to run the MosaStore storage system and the synthetic application. One node runs the MosaStore manager and 19 run the storage nodes, the MosaStore SAI, and application scripts.

Evaluation Results. Figure 2 and Figure 3 present the performance of our systems for small and medium workloads. The figures present distinctly (as stacked bars) the performance for data stage-in/out and the performance for the pipeline stages that touch the intermediate storage system. We experiment with four possible intermediate storages: (1) a local file system (labeled '*local'* in the plots) which represents the best possible performance and is presented as a baseline for comparison; (2) NFS itself used as intermediate storage (labeled '*NFS'* in the plots); (3) MosaStore applying standard configuration and optimization techniques (labeled '*MosaStore'*); and (4) a MosaStore with modifications to become workflow aware (labeled '*WFaware*').

For all scenarios, the workflow-aware system performs faster than NFS and MosaStore un-optimized, and is close to the performance of the local file system. The larger the file sizes, the larger the difference between the workflow-aware setup and the

other two alternatives of shared intermediate storage. For medium files, *WFaware* is 8x faster than NFS, and almost 2x faster than MosaStore. For large files (1GB), this difference is even larger, NFS is unable to properly handle the demand generated and we stopped the experiments after 200 minutes.

The *local* configuration presents the optimal data placement decision for the pipeline pattern. The *WFaware* storage system lags behind the local storage due to added overhead of metadata operations. We note that for larger file WFaware performance is close to the local performance as IO overhead dominates the execution time.

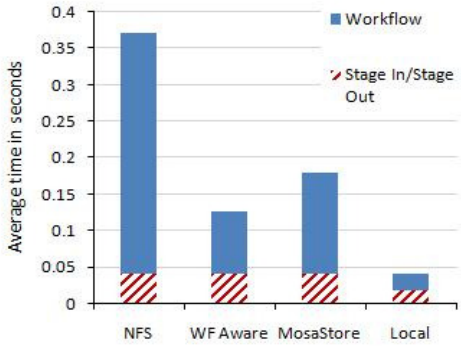


Figure 2. Pipeline pattern – small files. Average execution time (in seconds) for small file sizes.

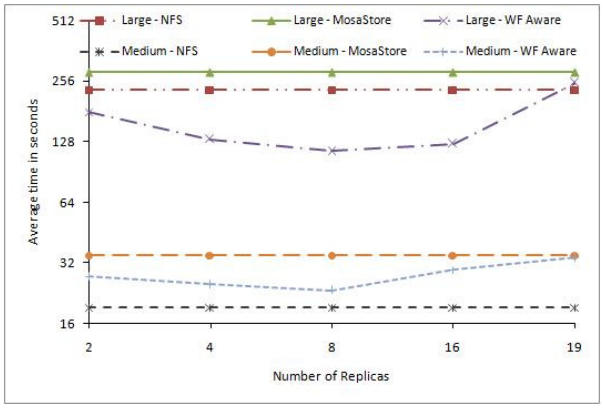
Figure 3.Pipeline pattern – medium files. Average execution time (in seconds) for medium-size file.

2) Broadcast Pattern Evaluation

Customization. To efficiently support the broadcast pattern for the workflow-aware system we added eager replication to the MosaStore base system (the system originally supported lazy replication only). With eager replication replicas are created in parallel, while a file is written to the storage system (if replication is needed for that file). A broadcasted file will be eagerly replicated by the storage system thus reducing the likelihood of a bottleneck when a file is consumed by multiple concurrent workflow stages.

The workload (Figure 1- center). One workflow stage produces a file into intermediate storage. The file is consumed by 19 executables running in parallel. Each of these writes its output independently to backend storage (*i.e.*, the NFS).

Evaluation Results. Figure 4 shows the performance for the broadcast pattern for all file sizes, while varying the number of replicas created. We omit workload small since its performance is almost constant across all systems and number of replicas (around 0.47 seconds). For medium and large files, the workflow-aware system performs faster than MosaStore (configured with no replication) and reaching the best performance for 8 replicas in both cases, this highlighting the potential benefits of workflow-awareness. For more than 8 replicas, the overhead of replicating the file is higher than the gain from adding more data access points. A similar pattern can be observed for small files; in this case, replication does not pay off.



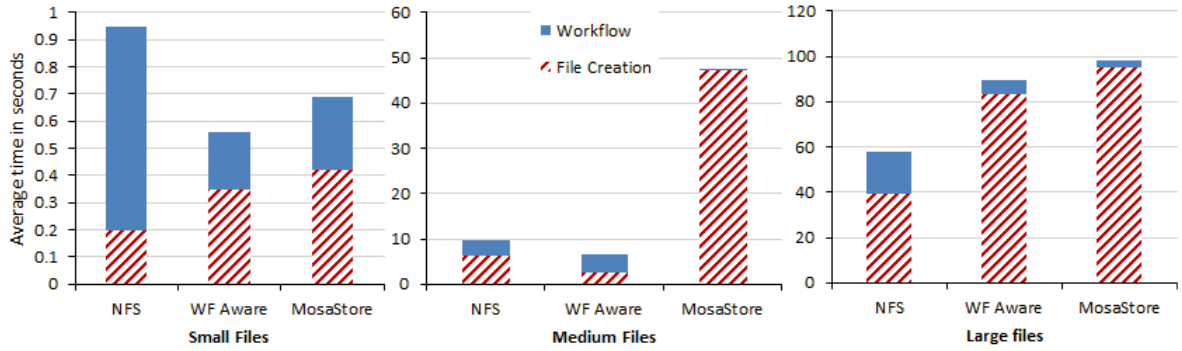


Figure 6. Reduce pattern. Average benchmark execution time (in seconds) for different file sizes and intermediate file systems.

Figure 4 - Broadcast pattern. Average time (in seconds) for different file sizes and intermediate storages. The workflow-aware storage system uses replication. Note that the y-axis uses logarithmic scale to present all results in one plot.

To offer a view that explains the trends Figure 5 shows the breakdown of the benchmark phases: creating the replicas and actual application runtime. As the number of replicas increases, the time to process the data (the 'workflow' line) decreases and the time to create the replicas increases, The lowest total runtime is reached for 8 replicas.

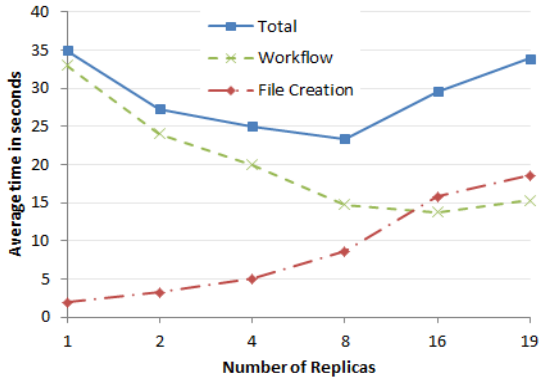


Figure 5 - Broadcast pattern phases. The figure presents runtime when running on top of the workflow-aware file system, for medium files and different replication levels.

We note that NFS performs the fastest for the medium size files. The reason, we believe is that the application benefits from: (i) relatively larger write buffers at the NFS server, and (ii) RAID in the server, which provides faster read/write throughput.

3) Reduce Pattern Evaluation

Customization. To efficiently support the reduce pattern, for workflow awareness we change the MosaStore data placement such that all output files of one stage are co-located on a pre-specified storage node. The synthetic application using the reduce pattern runs the reduce application on the nodes storing all the files increasing file access locality.

The workload(Figure 1 right). 19 executables running in parallel and produce intermediate files that are placed on intermediate storage and consumed by a single workflow stage which writes its output to the backend store (the NFS).

Evaluation Results. Figure 6 shows the performance for all three workloads and the tree ways to instantiate the intermediate storage systems (MosaStore, workflow-aware, and NFS used as intermediate storage). Similarly to the pipeline pattern, workflow-aware performs faster than NFS for small and medium file sizes, and better than MosaStore un-optimized in all scenarios. For medium files, workflow-aware is 8x faster than NFS and almost 2x faster than MosaStore default configuration. NFS performs faster than workflow-aware for large files: this happens because when placed simultaneously 19 large files overload the single disk on the single storage nodes in WFaware and MosaStore while NFS server with a 5 disks RAID and large memory is better provisioned to handle this workload.

IV. Summary and Future Work

This paper explores the viability of a workflow-aware storage system. To this end, we discuss the feasibility of building a workflow-aware storage system that can provide per-file optimizations, and evaluate experimentally the performance gains such system can bring. Our evaluation shows that a workflow-aware storage system can bring significant performance gains: up to 7x performance gain compared to a distributed storage system and up to 16x compared to a well provisioned NFS server.

Our future work goes in two directions: building a workflow-aware storage system to demonstrate in practice the potential gains this study highlights, and exploring solutions to determine, and communicate to the storage system, the application access patterns. We are currently exploring two approaches for determining application access pattern: by extending workflow compilers and runtime engines, that have full information about the workflow 'shape', to communicate file access patterns to the storage system through POSIX extended file attributes, and by building workload monitoring and access prediction

components and using predictions for storage system auto-tuning.

ACKNOWLEDGEMENTS

This research was sponsored by Department of Energy (DOE) and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC. In addition, we thank Ketan Maheshwari from University of Chicago for his comments on an earlier version of this paper.

REFERENCES

[1] *modFTDock*. [cited 2011; http://www.mybiosoftware.com/3d-molecular-model/922.

[2] A. C. Laity, N. Anagnostou, G. B. Berriman, J. C. Good, et al. *Montage: An Astronomical Image Mosaic Service for the NVO*. in *Proceedings of Astronomical Data Analysis Software and Systems (ADASS)*. 2004.

[3] Y. Chen, W. Chen, M. H. Cobb, and Y. Zhao, *PTMap A sequence alignment software for unrestricted, accurate, and full-spectrum identification of post-translational modification sites.* Proceedings of the National Academy of Sciences of the USA, 2009. **106**(3).

[4] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2004.

[5] I. Raicu, I. T. Foster, and Y. Zhao, *Many-Task Computing for Grids and Supercomputers*, in *IEEE Workshop on Many-Task Computing on Grids and Supercomputers* 2008.

[6] I. Foster, M. Hategan, J. M. Wozniak, M. Wilde, et al., *Swift: A language for distributed parallel scripting* Journal of Parallel Computing, 2011.

[7] E. Deelman, G. Singh, M.-H. Su, J. Blythe, et al., *Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems.* Journal of Scientific Programming, 2005. **13**(3): p. 219-237.

[8] E. Santos-Neto, S. Al-Kiswany, N. Andrade, S. Gopalakrishnan, et al. *Beyond Search and Navigability: Custom Metadata Can Enable Cross-Layer Optimizations in Storage Systems*. in *ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC) - Hot Topics Track*. 2008.

[9] J. Wozniak and M. Wilde. *Case studies in storage access by loosely coupled petascale applications*. in *Petascale Data Storage Workshop*. 2009.

[10] J. Bent, D. Thain, A. C.Arpaci-Dusseau, R. H. Arpaci-Dusseau, et al. *Explicit Control in a Batch-Aware Distributed File System*. in *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. 2004. San Francisco, California.

[11] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. *The Hadoop Distributed File System*. in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)* 2010.

[12] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. *The Case for Versatile Storage System*. in *Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 2009.

[13] I. Raicu, Y. Zhao, I. Foster, and A. Szalay, *Accelerating Large-scale Data Exploration through Data Diffusion*, in *International Workshop on Data-Aware Distributed Computing*. 2008.

[14] A. Chervenak, E. Deelman, M. Livny, M.-H. Su, et al., *Data placement for scientific applications in distributed environments* in *IEEE/ACM International Conference on Grid Computing*. 2007

[15] D. S. Katz, T. G. Armstrong, Z. Zhang, M. Wilde, et al., *Many-Task Computing and Blue Waters*, in *Technical Report CI-TR-13-0911. Computation Institute, University of Chicago & Argonne National Laboratory. arXiv:1202.3943v1*. 2012.

[16] T. Shibata, S. Choi, and K. Taura, *File-access patterns of data-intensive workflow applications and their implications to distributed filesystems*, in *International Symposium on High Performance Distributed Computing (HPDC)*. 2010.

[17] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, et al., *Characterization of Scientific Workflows*, in *Workshop on Workflows in Support of Large-Scale Science*. 2008.

[18] U. Yildiz, A. Guabtni, and A. H. H. Ngu, *Towards scientific workflow patterns*, in *Workshop on Workflows in Support of Large-Scale Science*. 2009.

[19] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, et al., *Minerva: An automated resource provisioning tool for large-scale storage systems.* ACM Transactions on Computer Systems (TOCS), 2001 **19**(4).

[20] L. B. Costa and M. Ripeanu. *Towards Automating the Configuration of a Distributed Storage System*. in *ACM/IEEE International Conference on Grid Computing (Grid)*. 2010.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. *The Google File System*. in *19th ACM Symposium on Operating Systems Principles*. 2003. Lake George, NY.

[22] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, et al., *GPFS-SNC: An enterprise storage framework for virtual-machine clouds* IBM Journal of Research and Development 2011.

[23] M. Rosenblum and J. K. Ousterhout. *The Design and Implementation of a Log-Structured File System*. in *ACM Transactions on Computer Systems*. February 1992.

[24] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, et al. *Ursa minor: versatile cluster-based storage*. in *FAST* 2005.

[25] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, et al. *Falkon: a Fast and Light-weight tasK executiON framework*. in *SuperComputing*. 2007.

[26] I. Raicu, I. Foster, Y. Zhao, P. Little, et al., *The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems*, in *International symposium on High Performance Distributed Computing (HPDC)*. 2009.

[27] Z. Zhang, D. Katz, M. Ripean, M. Wilde, et al. *AME: An Anyscale Many-Task Computing Engine*. in *Workshop on Workflows in Support of Large-Scale Science*. 2011.

[28] *FUSE, Filesystem in Userspace*. [cited 2011; http://fuse.sourceforge.net/.