A Cross-Layer Optimized Storage System for Workflow Applications

Samer Al-Kiswany[1]	Lauro B. Costa	Hao Yang	Emalayan Vairavanathan	Matei Ripeanu
University of Waterloo	Google Inc.	U. British Columbia	NetApp Inc.	Univ. of British Columbia
alkiswany@uwaterloo.ca	lbeltrao@gmail.com	haoy@ece.ubc.ca	emalayan@netapp.com	matei@ece.ubc.ca

Abstract— This paper proposes using file system custom metadata as a bidirectional communication channel between applications and the storage middleware. This channel can be used to pass hints that enable cross-layer optimizations, an option hindered today by the ossified file-system interface. We study this approach in the context of storage system support for large-scale workflow execution systems: Our workflow-optimized storage system (WOSS), exploits application hints to provide per-file optimized operations, and exposes data location to enable location-aware scheduling. We argue that an incremental adoption path for adopting cross-layer optimizations in storage exists, present the system architecture for a workflow-optimized storage system and its integration with a workflow runtime engine, and evaluate this approach using synthetic and real applications over multiple success metrics (application runtime, generated network stress, and energy). Our performance evaluation demonstrates that this design brings sizeable performance gains. On a large scale cluster (100 nodes), compared to two production class distributed storage systems (Ceph and GlusterFS), WOSS achieves up to 6x better performance for the synthetic benchmarks and 20-40% better application-level performance gain for real applications.

Keywords— Cross Layer optimizations, Distributed file systems, Workflow management, Batch processing systems

1 INTRODUCTION

Custom metadata features (a.k.a., 'tagging') have seen increased adoption in systems that support the storage, management, and analysis of 'big-data'. However, the benefits expected are all essentially realized at the application level either by using metadata to present richer or differently organized information to users (e.g., enabling better search and navigability [1, 2]) or by implicitly communicating among applications that use the same data items (e.g., to support provenance, or inter-application coordination).

***Our thesis** is that*, besides the above uses, *custom metadata can be used as a bidirectional communication channel between applications and the storage system* and thus become the key enabler for cross-layer optimizations that, today, are hindered by an ossified file-system interface.

This communication channel is *bidirectional* as the cross-layer optimizations enabled are based on information passed in both directions across the storage system interface (i.e., application to storage and storage to application). Possible cross-layer optimizations include:

- (*top-down*) Applications can use metadata to provide hints to the storage system about their future behavior, such as: per-file access patterns, ideal data placement (e.g., co-usage), predicted file lifetime (i.e., temporary files vs. persistent results), access locality in a distributed setting, desired file replication level, or desired quality of service. These hints can be used to optimize the storage layer.
- (*bottom-up*) The storage system can use metadata as a mechanism to expose key attributes of the data items stored. For example, a distributed storage system can provide information about data location, thus enabling location-aware scheduling.

The approach we propose has four interrelated advantages: it uses an application-agnostic mechanism, it is incremental, it offers a low cost for experimentation, and it focuses the research community effort on a single storage system prototype, saving considerable development and maintenance effort dedicated, nowadays, to multiple storage systems each targeting a specific workload (e.g., HDFS and PVFS [3]). First, the communication mechanism we propose: simply annotating files with arbitrary <*key, value*> pairs, is application-agnostic. Second, our approach enables evolving applications and storage-systems independently while maintaining the current interface (e.g., POSIX), and offers an incremental transition path for legacy applications and storage-systems: A legacy application will still work without changes (yet will not see performance gains) when deployed over a new storage system that supports cross-layer optimizations. Similarly a legacy storage will still support applications that attempt to convey optimization hints, yet it will not offer performance benefits. As storage and applications incrementally add support for passing and reacting to optimization hints, the overall system will see increasing gains. Finally, exposing information between different system layers implies tradeoffs between performance and transparency. To date, these tradeoffs have been scarcely explored. We posit that a flexible encoding (key/value pairs) as the information passing mechanism offers the flexibility to enable low-cost experimentation within this tradeoff space.

The approach we propose falls in the category of 'guided mechanisms' (i.e., solutions for applications to influence data placement, layout, and lifecycle), the focus of other projects as well. In effect, the wide range (and incompatibility) of past solutions proposed in the storage area in the past two decades (and incorporated to some degree by production systems – pNFS, PVFS [3], GPFS [4], Lustre, and other research projects [5-7]), only highlights that *adopting a unifying abstraction is an area of high potential impact*. The novelty of this paper comes from the "elegant simplicity" of the solution we propose. First, unlike past work, we maintain the existing API (predominantly POSIX compatible), and, within this API, we propose using the existing extended file

[1]Corresponding author. Address: David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, Canada N2L 3G1
E-mail address: alkiswany@uwaterloo.ca

attributes as a flexible, application-agnostic mechanism to pass hints across the application/storage divide. Second, and equally importantly, we propose an extensible storage system architecture that can be extended with application specific optimizations.

We demonstrate our approach by building a POSIX-compatible storage system to efficiently support one application domain: scientific workflows (an application domain detailed in §2 and Figure 1). We chose this domain as this community has to support a large set of legacy applications (developed using the POSIX API). Our storage system is instantiated on-the fly to aggregate the resources of the computing nodes allocated to a batch application (*e.g.*, disks, SSDs, and memory) and offers a shared file-system abstraction with two key features. First, it optimizes the data layout (e.g., file and block placement, file co-placement) to efficiently support the workflow data access patterns (as hinted by the application). Second, the storage system uses custom metadata to expose data location information so that the workflow runtime engine can make location-aware scheduling decisions. These two features are key to efficiently support workflow applications as their generated data access patterns are irregular and application-dependent.

Contributions. This project demonstrates that it is feasible to have a POSIX compatible storage system that can be yet optimized for each application (or application mix) even if the application has a different access pattern for different files. The key contributions of this work are:

- *We propose a new approach* that uses custom metadata to enable cross-layer optimizations between applications and the storage system. Further, we argue that this approach can be adopted incrementally. This suggests an evolution path for co-designing POSIX-compatible file-systems together with the middleware ecosystem they coexist such that performance efficiencies are not lost and flexibility is preserved, a key concern to support legacy applications.
- *We present an extensible storage system architecture* that supports cross-layer optimizations. We demonstrate the viability of this approach through a storage system prototype optimized for workflow applications (dubbed WOSS, Figure 1). WOSS supports application-informed data placement based on per-file hints, and exposes data location to enable location-aware task scheduling. Importantly, we demonstrate that it is possible to achieve our goals, with only minor changes to the workflow scheduler, and without changing the application code or tasking the developer to annotate their code to reveal the data usage patterns.
- *We demonstrate, using synthetic benchmarks as well as three real-world workflows, that this design brings sizeable performance gains*. On a large scale cluster (100 nodes), compared to two production class distributed storage systems (*Ceph* [8] and *GlusterFS* [9]), WOSS achieves up to 6x higher performance for the synthetic benchmarks and 20-40% application-level performance gain for real applications.

Organization of this paper. The final section of this paper includes a detailed design discussion and design guidelines, discusses the limitations of this approach, and elaborates on the argument that custom metadata can benefit *generic* storage systems by enabling cross-layer optimizations (§5). Before that, we present the context (§2), the design (§3) and evaluation (§4) of a first storage system we designed in this style: *the workflow-optimized storage system (WOSS).*

2 BACKGROUND AND RELATED WORK

This section starts by briefly setting up the context: the target application domain and the usage scenario. It then continues with a summary of data access patterns of workflow applications (§2.1) and a survey of related work on alleviating the storage bottleneck (§2.2).

The application domain: workflow applications. Meta-applications that assemble complex processing workflows using existing applications as their building blocks are increasingly popular in the science domain [10-13]. A popular approach to support these workflows is *the many-task approach* [14], through which the workflow assembles a set of independent processes that communicate through intermediary files stored on a shared POSIX file-system (e.g., Montage workflow detailed in Figure 15). The dependency between the different executables in a workflow are expressed in scripts or special domain-specific languages [15].

The workflow scheduler schedules the workflow tasks on a set of compute nodes allocated exclusively to the workflow (Figure 1). The scheduler submits the tasks only when all its input files are available in the storage system. Workflow tasks access the storage system through the Linux file system API.

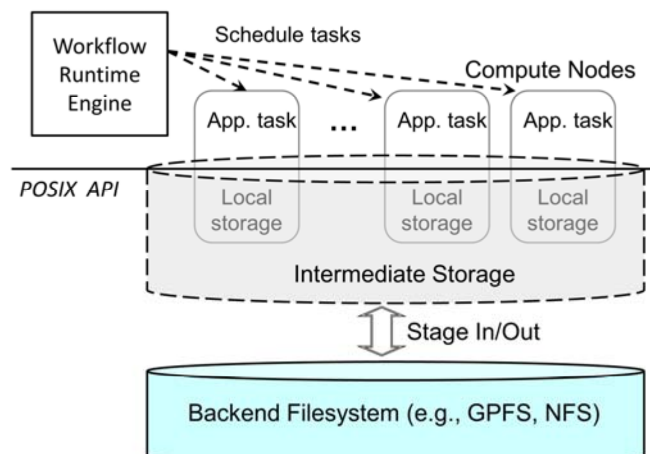

Figure 1. **Usage scenario and high-level architecture.** The intermediate storage aggregates the storage space of the compute nodes and is used as an intermediate file-system. Input/output data is staged in/out from the backend storage. The scheduler submits tasks to individual compute nodes.

Three main advantages make most workflow runtime engines adopt this approach: simplicity, direct support for legacy applications and support for fault-tolerance. First, a shared file-system approach simplifies workflow development, deployment and debugging: essentially workflows can be developed on a workstation then deployed on a large machine without changing the environment. Moreover, a shared file-system simplifies workflow debugging as intermediate computation state can be easily inspected at runtime and, if needed, collected for debugging or performance profiling. Second, a shared file-system supports the legacy applications that form the individual workflow stages as these generally use the POSIX API. Finally, this approach simplifies fault-tolerance: a failed execution step can simply be restarted on a different compute node as long as all its input is available in the shared file-system.

Although these are important advantages, the main drawback of this approach is low performance: the file-system abstraction constrains the ability to harness performance-oriented optimizations that can only be provided if information is shared between system layers (Figure 1). More specifically, a traditional file system cannot use the information available at the level of the workflow execution engine (e.g., to guide the data placement). Similarly, as traditional file-systems do not expose data-location information, the workflow runtime engine cannot exploit opportunities for collocating data and computation.

Usage scenario: batch applications. Since, on large machines, the back-end file-system becomes a bottleneck when supporting I/O intensive workflows [16], today's common way to run them is to harness some of the resources allocated by the batch-scheduler to the application and assemble a scratch shared file-system that will store the intermediary files used to communicate among workflow tasks. This is batch-oriented scenario that we assume for the rest of the paper (described in detail in Figure 1).

Table 1. Common workflow patterns. Circles represent computations, an outgoing arrow indicates that data is produced (to a temporary file) while an incoming arrow indicates that data is consumed (from a temporary file), rectangles represent blocks of a file. There may be multiple inputs and outputs via multiple files. Arrows are labeled with extended attribute API calls used to pass hints to enable the optimizations. The corresponding hints are presented in detail in Table 3.

Pattern Details	Optimizations / Hint
Pipeline Input set(dp_local) get(file, location) set(dp_local) get(file, location) Output	▪ Node-local data placement (if possible). ▪ Caching. ▪ Data location-aware scheduling.
Broadcast Input set(replicate, 4) get(file, location) Output	▪ Optimized replication taking into account the data size, the fan-out and the network topology.
Reduce Input set(dp_collocate) get(file, location) Output	▪ Reduce-aware data placement: co-placement of all output files on a single node; ▪ Data location-aware scheduling
Scatter Input file set(block-size, size) set(dp_striped) get(file, location) Output	▪ Application-informed block size for the file. ▪ Application-aware block placement; ▪ Data-location application scheduling
Gather Output file	▪ Application-informed block size for the file. ▪ Application-aware block placement;
Distribute Input Output files	▪ Application informed file and block placement. ▪ Application informed replication.

2.1 Common Workflow Data Access Patterns

To make this paper self-contained, this subsection briefly presents the common data usage patterns identified by past studies [10-13], the opportunities for optimizations they generate, and the required storage system support. We detail next only a few of the patterns:

- *Pipeline*: A set of compute tasks are chained in a sequence such that the output of a task is the input of a next task in the chain. An optimized system stores an intermediate output file on the same storage node on the same machine that executes the task that produced the file (if space is available) to increase access locality and efficiently use local caches through maintaining the cache content between tasks until the file is consumed. Ideally, the location of the data 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 used by multiple tasks. 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.
- *Reduce*: A single task uses input files produced by multiple computations. An optimized storage system can place all these input files on one node and expose their location, thus creating an opportunity for scheduling the reduce task on that node to increase data access locality.
- *Scatter*: Disjoint 'regions' of a single file are read by multiple tasks that can be executed in parallel. An optimized object storage system can configure the block-size (i.e., the internal storage unit) to be smaller than the region size (such that no two regions share a block) and optimize its operation by placing all the blocks that belong to a particular region on the same node, then placing different file regions on different nodes. Exposing block-level data placement information will enable location aware scheduling.
- *Gather*: A single file is written by multiple compute nodes where each node writes to a disjoint region of the file. An optimized storage system can configure the block size to be smaller than that the region size (such that no two regions share a block) and provide support enabling parallel writes to all file regions. The data placement is then optimized based on the next step in the workflow (e.g. written on a single node for pipeline, or on multiple nodes for scatter).
- *Distribute*: A collection of files is generated by a task is consumed by multiple tasks running on different compute nodes. An optimized storage system can use a smart data placement scheme to support the distribute pattern. Such storage will spread the files in a balanced way across storage nodes.

2.2 Past Work on Alleviating the Storage Bottleneck

1) 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 [17] optimizes for large datasets and append access, HDFS and GPFS-SNC [18] optimize for immutable data sets, location-aware scheduling and rack-aware fault tolerance; the log-structured file system [19] optimizes for write-intensive workloads, ROMIO [20] optimizes for MPI application access pattern; Finally, PVFS [3], optimizes for parallel file access. 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, as in the case of workflows.

2) Dealing with a constraining storage system interface. Two solutions are generally adopted to pass hints from applications to the storage system: either giving up the POSIX interface for a wider API, or, alternatively, extending this API with an orthogonal additional ad-hoc interface for hint passing. Most storage systems that operate in the HPC space (pNFS, PVFS, GPFS, Lustre) fall in the latter category; while most Internet services/cloud storage systems (e.g., HDFS) fall in the former category. In terms of exposing data location, the situation is similar: HDFS and other non-POSIX systems do expose data location to applications while most parallel large-scale file systems (e.g., pNFS, PVFS, GPFS) do not. Further we note that these systems cannot support cross-layer optimizations as their design does not support per-file optimizations, does not have mechanisms to enable/disable optimizations based on application triggers, or it does not allow extending the system.

Table 2. Survey of related projects. The table compares WOSS with current approaches on number of axes

Projects	Domain specific / general	Production / research project	API	Bidirectional	Extensible
HDFS, ROMIO [20], ADIOS [21]	Domain	Production	New API	N	N
PVFS [3], GPFS [4]	Domain	Production	POSIX	N	N
GreenStor [22], eHiTS [23], UrsaMinor [24],	General	Research	New API	N	N
Mesnier et. al. [25]	General	Research	Modified Disk API	N	N
BitDew [5]	Domain	Research	New API	N	N
WOSS	Domain	Research	POSIX	Y	Y

3) Storage system optimizations using application provided hints (Summarized in Table 2). A number of projects propose exploiting application information to optimize storage system operations. ADIOS [21] proposes a new API that allows the developers to select different implementations for the IO routines. Mesnier et al. [25] propose changes to the storage block API to classify blocks into different classes (metadata, journal, small/large file), allowing the storage system to apply per class QoS polices. Patterson et al. [26] propose an encoding approach to list the blocks an application accesses, which the storage system uses to optimize caching and prefetching. eHiTS [23] and GreenStor [22] propose using application hints to facilitate turning off or idling storage devices holding 'cold' data blocks. BitDew [5], a programming framework for desktop grids, enables users to provide hints related to desired data reliability, and suggest optimal data transfer protocol. Finally, UrsaMinor storage system [24] allows the system admin or the application to configure, per object, the reliability mechanism.

Recently, a number of systems used extended attributes to enable new file system operations. For instance, SELinux, and Solaris ZFS [27] use extended attributes to store the file access control list, GPFS [4] use them to allow controlling the file replica-

tion and caching policy, while OrangeFS [28] uses extended attribute to specify the stripe width or distribution parameters for a directory.

These efforts differ from our proposed approach in three main ways (summarized in Table 2): First, most of them target a specific optimization. Second, they propose unidirectional hint passing from application to storage. Third, they either propose changes to the standard APIs to pass hints, or use the current API in a non-portable way. This hinders adoptability and portability. Finally, to the best of our knowledge, no other project provides a bidirectional communication mechanism, and proposes an *extensible* storage system design that can be extended to support new optimizations, while using a standard API. We note that, theoretically, all storage system optimizations proposed in the surveyed projects can use our cross-layer optimization approach, and can be implemented using our architecture.

4) Monolithic workflow runtime engine design. A number of projects (e.g., Falkon [29, 30], AME [31]) give up the layered design and go for a monolithic design that couples the workflow runtime engine and the data-store. The scheduler will keep track of data location and will use this information to submit jobs where data is located or orchestrate explicit data moves so that data is available on the local storage of a node when a task starts. While this approach may to lead to higher performance (an observation that holds for all monolithic designs) we believe that its drawbacks are non-negligible: loss of transparency, higher system complexity, and inability 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 and the shared file system approach. WOSS strikes a balance between these two approaches, it enables optimizing the data movement during the workflow execution, without sacrificing transparency and layering, and without increasing system complexity.

3 System Architecture

To efficiently support the usage scenario targeted and the access patterns generated by workflow applications (§2.1), the WOSS design needs to *provide per-file (or group of files) runtime configurability* to support the diverse data-access patterns different workflow stages may have, and needs to be *extensible:* that is, to allow defining new optimizations and associate them with new custom attributes. This section presents the system design (§3.1) and integration with the workflow scheduler (§3.2) while focusing on how to provide the two aforementioned properties (§3.3 and §3.4), our prototype implementation (§3.5), and the integration details with *pyFlow* and *Swift,* the two workflow runtimes we experiment with (§3.6).

3.1 System Design

WOSS is based on a traditional object-based distributed storage system architecture with three main components (Figure 2): a centralized metadata manager, the storage nodes, and the client's system access interface (SAI). This architecture is indeed widely adopted today (e.g., GFS, PVFS, Lustre).

- The metadata manager maintains the entire system metadata (e.g., storage node status, block-mapping for each file , and file metadata). Similar to a number of other storage systems the current implementation of WOSS adopts a centralized metadata manager implementation.
- The storage nodes contribute storage space to the system. To facilitate integration, the design aims to decouple, to the extent possible, the metadata manager and the benefactor nodes, and in effect minimizes the set of functions storage nodes provide. Storage nodes interact with the manager to publish their status (on-/off-line) and free space using soft-state registration, serve client requests to store/retrieve data blocks, and run garbage collection algorithms.
- The storage access interface (SAI) which provides the client-side POSIX file system interface.

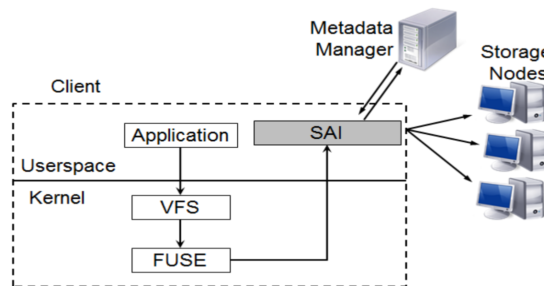

Figure 2.The main components of WOSS: the metadata manager, the storage nodes and the client module: SAI (detailed here, implemented on top of FUSE)

Each file is divided into fixed-size blocks that are stored on the storage nodes. Data storage and retrieval operations are initiated by the client via the SAI. To retrieve a file, the SAI first contacts the metadata manager to obtain the block-map, (i.e., the location of all blocks corresponding to the file), then, the actual transfer of data blocks occurs directly between the storage nodes and the SAI. When a new file is written, the SAI first contacts the metadata manager to allocate storage space on the storage nodes. Once the space is allocated the SAI will proceed with storing the new blocks on the storage nodes. As the SAI cannot predict the file size in advance, it allocates the space incrementally. If this space is not used, it is asynchronously garbage collected.

3.2 Integration with a Workflow Runtime System

Figure 3 details the WOSS integration with the workflow scheduler. The workflow scheduler passes to the WOSS metadata manager file access hints. These hints are used to optimize specific optimization for each file. Further, the workflow schedule queries the WOSS metadata manager for file location information and uses this information to perform location-aware scheduling. The scheduler and workflow tasks access the system through the Linux file system API implemented by the SAI.

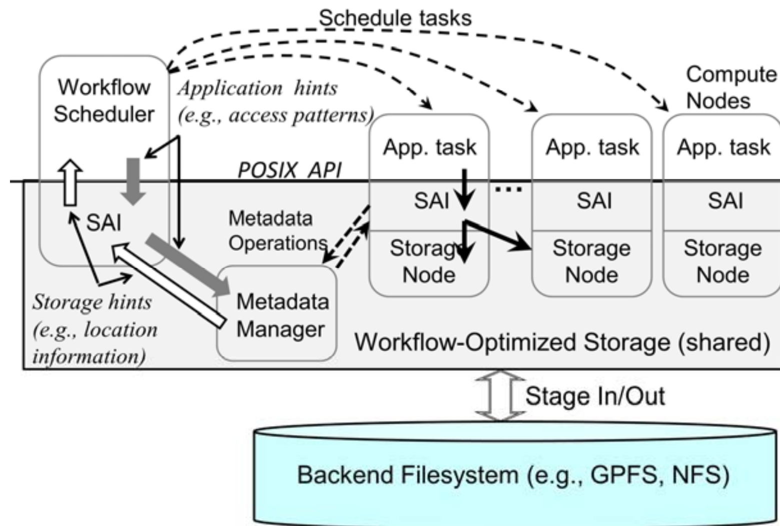

Figure 3. **System Architecture.** The workflow optimized storage system (WOSS) aggregates the storage space of the compute nodes and is used as an intermediate file-system. Input/output data is staged in/out from the backend storage. The workflow scheduler provides hints about a file's access pattern to the metadata manager. These hints are used to optimize WOSS operations. The workflow scheduler queries WOSS metadata manager for data location to preform location-aware scheduling. The scheduler and application tasks access WOSS through the system access interface (SAI) using the Linux file system API.

3.3 Design for Per-File Configurability

The distributed nature of the storage system makes providing hint-triggered optimizations challenging: while the hints (i.e., the file's extended attributes) are maintained by the manager, the functionality corresponding to the various optimizations can reside at the manager (e.g., for data placement), client SAI (e.g., for caching) or storage nodes (e.g., for replication). This section details the design features that allow per-file optimizations, while the next section details the design features that enable extensibility.

To allow external control of per-file optimizations the three WOSS components adopt a dispatcher-based design. Figure 4 details the design of the metadata manager. The storage node and SAI follow the same design. In this design all requests received by the component are processed by a dispatcher. The dispatcher uses hints provided by the scheduler to select among multiple implementation of the requested operation. For every storage operation the system provides a generic default implementation, if no hints are provided for a file the dispatcher will forward the request to the default implementation of the operation.

Figure 4 shows one complete example. The scheduler indicates that the file "f.dat" will have a "local" (pipeline) access pattern (step 1 in Figure 4). This information is stored in the file metadata. When an SAI asks for an allocation for file "f.dat" (step 2) the dispatcher queries the file metadata (step 3) and forwards the call to the allocation module optimized for "local" pattern (step 4). The module will allocate the storage space and respond to the client SAI (step 5). The client uses the allocation to store the new file on the allocated storage node (step 6).

When the scheduler wants to schedule a task that consumes "f.dat", the scheduler will query the metadata manager for file location (step 7). The dispatcher will forward the request to the file location module which will respond to the scheduler (step 11). The scheduler will use this information to schedule the task on the node that has the file (step 12).

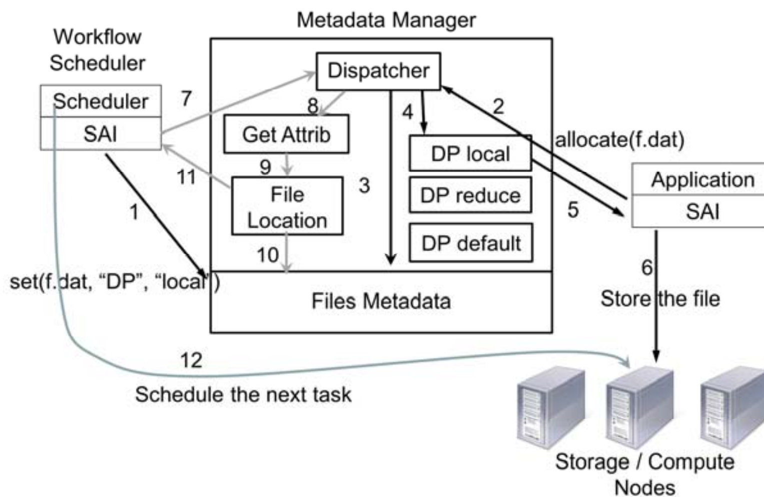

Figure 4. System Design. WOSS component adopt a dispatcher-based desig in which every operation is dispatched to different module based on scheduler-provided hints. The figuer details the metadata manager design, the SAI and storage nodes follow the same design. Black lines show the steps for providing hints and writing a new file, while gray lines show the steps for scheduling a task based on location-aware scheduling.

3.4 Design for Extensibility

In addition to enabling per-file optimization, we aim for a flexible design that supports exploration and facilitates adding new custom metadata and their corresponding optimization. The distributed nature of the system requires having an extensible design for components as well as the communication protocols connecting these components.

To this end, three key design decisions enable extensibility:

- *Extensible storage system components design*. All storage system components follow a dispatcher-based design (Figure 4). The design allows extending the components with additional versions of storage system operations (e.g., new space allocation, replication, or write operation) and specifying the hint that will trigger that implementation.
- *A generic inter-component hint propagation approach* that extends every message/request with optimization hints to enable propagating hint information between components (manager, storage node, SAI). This enables end-to-end information passing and optimized handling of every message/request across components.

  In our design, file-related operations are always initiated by the client SAI (e.g. space allocation, data replication request, or checking file attributes,). The first time an application opens a file or gets the file attributes, the SAI queries the metadata manager and caches the file's *extended* attributes (that carry the application hints). The SAI tags all subsequent inter-component communication related to that file (e.g., a space allocation, a request to store a data block) with the file's extended attributes. These hints will trigger the corresponding callbacks at each component.
- *Passing hints bottom-up:* To communicate a storage hint to the application, the metadata manager provides an extensible information retrieval module (the GetAttrib module in Figure 4). This module is integrated with the dispatcher described in the previous section as it is only triggered by the client POSIX 'get extended attribute' operation. Similar to other optimizations, to extend the system to expose specific internal state information, the developer needs to decide the application hint/tag (key-value pair) that will trigger the optimization. The module, as all other optimization modules, has access to the manager metadata and the system status information, and is able to extract and return any internal information.

To extend the system with a new optimization for a specific operation (e.g., space allocation, replication, read, or write), the developer needs to decide the application hint (key-value pair) that will trigger the optimization, and implement the callback function on all system components related to the optimization. Every optimization module can access the storage component's internal information, including the manager metadata or system status (e.g., storage nodes status), and access the blocks stored at the storage nodes, through a well-defined API. For optimizations implemented on multiple components, no special communication between components is needed as the hints piggyback on the inter-component messages. Finally, the developer of the new optimization module has the flexibility of persisting optimization-special data on disk, or forking new threads. While it is hard to quantitatively measure the architecture's extensibility, we have verified that it is possible to implement all storage system optimizations surveyed in the related work (§2.1) or proposed in §5, and can provide the information to enable all application level optimizations surveyed in §5.

3.5 Prototype Implementation Details

We based our prototype implementation on MosaStore (*www.mosastore.net*) an object-based distributed storage system. Our prototype changes MosaStore design and implementation to follow the extensible storage system design as described above. WOSS SAI uses FUSE kernel module to provide the POSIX file system interface.

We highlight a number of implementation details:

- *Replication operations* are carried by the storage nodes. Their design adopts a similar dispatcher architecture to enable multiple replication policies. In the current implementation, the application can select the replication policy and the number of replicas. Two replication policies are available: eager parallel replication (to replicate hot spot files as used in the broadcast pattern), and lazy chained replication (to achieve data reliability without increasing system overhead).
- *Exposing data location*: To expose file's location, our system defines a reserved extended attribute that has values for every file in the system ("location"). An application (in our case the workflow runtime) can 'get' the "location" extended attribute to obtain the set of storage nodes holding the file.
- *Data placement*. The prototype implements a set of data placement strategies (Table 3) each optimized for a specific application access pattern (§2).

Table 3. Implemented metadata attributes (hints) and the corresponding optimizations. The left column presents an example of the API call. *DP* stands for Data Placement,

Hints (name, value)	Pattern	Effect Location	Description
DP, Local	Pipeline	Metadata, SAI, Storage nodes	Indicates preference to allocate the file blocks on the local storage node
DP, collocation | <*group_name*>	Reduce	Metadata, Storage nodes	Preference to allocate the blocks for all files within the same <*group_name*> on one node.
DP, scatter | <*scatter_size*>	Scatter	Metadata, Storage nodes	Place every group of contiguos <*scatter_size*> blocks on a storage node
Replication, <*num_replicas*>	Broadcast	Metadata, Storage nodes	Replicate the blocks of the file <*num_replicas*> times.
ReplicationSemantics, *semantic*	Broadcast and Replication	Storage nodes	Indicates which replication semantic to be used for the file: *optimistic*, return to application after creating the first replica, *pessimistic*, return to the application only after a block is well replicated.
CacheSize, <*size*>	Pipeline	SAI	Suggest a cache size per file (e.g. small cache size for small files or for read once files)
BlockSize, <*size*>	Scatter, Gather, Random file access	SAI, Storage nodes	Suggest a block size per file (e.g., scatter, gather, large blocks for sequential read/write files, small blocks for random read files)
Location	All	Metadata, Scheduler	*Retrieves* the location information of the specific file.

Prototype limitations. The prototype has two main limitations (all these limitations are the result of implementation decisions that enabled faster development and can be easily addressed with more development resources). First, the data placement tags are only effective at file creation, changing the data placement tag for existing files will not change the file layout. Second, our prototype uses a centralized metadata manager. While this introduces a potential scalability bottleneck, our experience is that the bottleneck that limited the overall system performance lay with the workflow runtime engine rather than with the storage system. We note that nothing in our design precludes adopting a more scalable multi-node metadata manager design.

3.6 Integration Details

To demonstrate the end-to-end benefits of our approach, we integrated the WOSS prototype with *Swift*, a popular language and workflow runtime system [32] and *pyFlow*, a similar, yet much simpler, runtime we have developed ourselves (we stress that our integration does not require any modification to the application tasks). In particular we applied two modifications:

- *Adding location-aware scheduling*. The current *Swift* and *pyFlow* implementations do not provide location-aware scheduling. We modified the schedulers to first query the metadata manager for location information, then attempt to schedule the task on the node holding the file. We note that our scheduling heuristics are relatively naïve, we estimate that further performance gains can be extracted with better heuristics; thus, our experiments provide a lower bound on the achievable performance gains.
- *Passing hints to indicate the data access patterns.* Our experiments assume that the workflow runtime engine performs the task of determining the data access pattern because we consider it the most direct approach to obtain this information. The reason is that the runtime engine has access to the workflow definition, maintains the data dependency graph, and uses them to schedule computations. Thus, it already has the information to infer the usage patterns, the lifetime of each file involved, and make computation placement decisions. Changing the workflow runtime implementation, however, to automatically extract this information is a significant development task (and not directly connected with the thesis we put forward here). Thus, we take a simpler approach: we inspect the workflow definitions for the applications we use in our evaluation, and explicitly add the instructions (2 to 5 *setattrib*() calls depending on the workflow) to indicate the data access hints.

Integration limitations: As one of our experiments highlights, our integration of location-aware scheduling with *Swift* adds a significant overhead. This, for some scenarios at scale, eliminates the performance gains brought by our optimizations. The problem is that, to limit the changes we make in the *Swift* code, we currently implement every set-tag or get-location operation as a *Swift* task which, in turn, needs to be scheduled and launched in a computing node to call the corresponding setattrib/getattrib command. With more time we can integrate this with *Swift*'s language and its Java-based implementation. The corresponding overhead is evaluated in §4.3 with *pyFlow.*

4 EVALUATION

We have deployed MosaStore on a wealth of different platforms (e.g., from vanilla Linux clusters, to a BlueGene/P machine, to virtualized platforms like Grid5000 and Amazon EC2 - within one region and across multiple regions). We have also run a large number of workflow applications. This section summarizes our experience so far, more details are in our technical report [33].

The section is structured as follows: we first present the context of our comparison, then present the performance results when using synthetic workloads (§4.1), simple application workflows – BLAST and modFTDock (§4.2), and finally we offer a detailed performance analysis when running Montage - a complex 10-stage workflow at large scale (100 nodes) (§4.3).

The storage systems we compare with. We compare the proposed workflow-optimized storage system (labeled ***WOSS*** in the plots) with the following alternatives:

- First, we compare WOSS with a number of other intermediate storage systems deployed over node-local spinning disks as well as over RAM-disks: *(i)* We use *MosaStore* without any cross-layer optimizations as this is the performance we expect form a traditional object-based **d**istributed **s**torage **s**ystem design, hence we name this deployment ***DSS*** from now on. *(ii)* Additionally, for experiments on the Grid 5000 platform, we also compare with *Ceph* [8] and *GlusterFS* [9]. Since these systems are similar in terms of architecture and design to other widely adopted storage systems, such as Lustre and PVFS [3], comparing WOSS against them gives a rough estimate of the potential performance gains our technique can enable. In the past [34] we have also showed that WOSS compares favorably against Chirp[35]/Parrot[36] when deployed in similar conditions as intermediate storage.
- Second, we use a typical backend persistent storage (e.g., GPFS or NFS) available on clusters and supercomputers as another baseline. The reason for this additional baseline is to estimate the gains brought by the intermediate storage scenario and, additionally, to show that DSS is configured for good performance. In the past [34] we have also shown that WOSS compares favorably against a number of other backend storage options available for the Amazon cloud: S3 and ECB (both accessed through FUSE based clients similar to our system).
- Finally, where possible, we use a third baseline: node-local storage. This baseline exposes the *optimal* achievable performance.

Workload: applications and synthetic benchmarks. In §4.1 we summarize the results for synthetic benchmarks and show detailed results for one large scale execution of the pipeline benchmark. We then (§4.2 and §4.3) present results for three real applications that demonstrate the performance benefits of the proposed approach.

Integration with workflow runtime engines. To demonstrate that our approach and implementation are application-, workflow engine-, and platform-agnostic and that they bring performance improvements in multiple setups the synthetic benchmarks are implemented solely using shell scripts and *ssh* while the real applications use two workflow execution engines: *pyFlow* or *Swift*. These engines and the shell scripts query the storage system for file location before launching a task on a specific machine.

Testbeds. We use two testbeds. The first testbed (TB20) is our lab cluster with 20 machines. Each machine has Intel Xeon E5345 4-core, 2.33-GHz CPU, 4-GB RAM, 1-Gbps NIC, and a RAID-1 on two 300-GB 7200-rpm SATA disks. The system has an addi-

tional NFS server as a backend storage solution that runs on a better 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. The NFS provides backend storage to the applications.

The second testbed (TB100), used for larger scale experiments, includes 100 nodes on Grid5000 'Nancy' cluster (www.grid5000.fr; [37]). Each machine has Intel Xeon X3440 4-core, 2.53-GHz CPU, 16-GB RAM, 1-Gbps NIC, and 320GB / SATA II. The NFS server runs on a Inter Xeon L5640 2.27Ghz, 24 cores and 48GB of RAM using RAID 1 on 600GB/SATAIII/15K RPM.

Deployment details. When evaluating WOSS and DSS systems, one node runs the metadata manager and the workflow coordination scripts, while the other nodes run the storage nodes, the client SAI, and the application executables.

Where possible we present performance results for two deployments of all the intermediate storage systems we compare (WOSS, DSS, Ceph, Gluster, node-local): deployed over RAM-disks and spinning-disks.

Gluster and Ceph configurations are set to be similar to WOSS and DSS: they run clients and storage modules on all nodes available to the application except on one node where the storage manager is running. Ceph and GlusterFS deployment scripts are provided by the Grid5000 platform and shared by all users, thus we assume that administrators configure them for good performance. For WOSS and DSS, we use the default configurations that have proved to work reasonably well for most situations, although do not provide optimal performance for all deployments and applications.

We note that, WOSS performance was also compared to a set of cloud storage systems (e.g., HDFS, Amazon S3) [34]. WOSS achieved the best performance for both synthetic and real applications workloads for most of the configurations.

Success metrics. We use multiple success metrics: application runtime (for all experiments) as well as generated network stress, and consumed energy.

Further, we have compared WOSS with DSS in terms of energy consumption [38] with synthetic benchmarks and real application. WOSS reduces the energy consumption by up to 50% while significantly improving the system performance.

4.1 Synthetic Benchmarks

The synthetic benchmarks provide relatively simple scenarios that highlight the potential impact of cross-layer optimizations in an intermediate storage scenario for each of the patterns used. These benchmarks (presented in Figure 5) are designed to mimic the application access patterns described in §2.1.

Staging-in/out: Current workflow systems generally use an intermediate storage scenario: they stage-in input data from a backend store (e.g., GPFS) to the intermediate shared storage space, process the data in this shared space, and then stage-out the results to be stored on the backend storage. Overall, WOSS and DSS perform faster than NFS for the staging time. Although this section does not target evaluating staging, it reports stage-in/out for the actual benchmark separately from the workflow time. *Note that adding the staging to the benchmark is conservative: these patterns often appear in the middle of an application workflow, hence staging would not typically affect them.*

Pipeline benchmark. Each pipeline stages-in a common input file from the shared backend (i.e., the NFS server), goes through three processing stages using the intermediate storage, then the final output is staged out back to backend. The script tags the output files produced by a pipeline stage with a *'local'* tag to inform the storage to attempt storing the output files on the node where the task runs. WOSS exposes the file location so that the benchmark can launch the next stage on the machine that already stores the file.

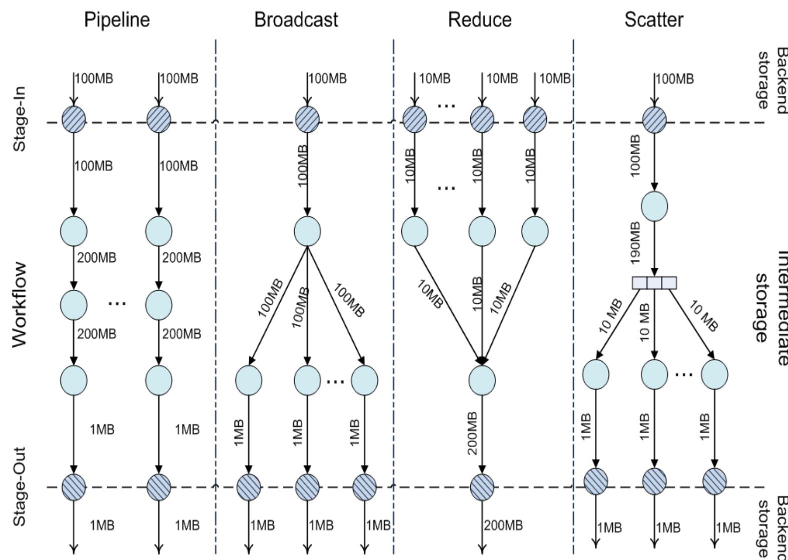

Figure 5. Pipeline, broadcast, reduce, and scatter synthentic benchmarks. Nodes represent workflow stages (or stage-in/out) and arrows represent data transfers through files. Labels represent file sizes. Horizontal dashed lines represent the crossing boundary between backend and intermediate storage (e.g., stage-in reads a file from the backend and writes to the intermediate storage). We also executed the benchmarks with data sizes scaled up (10x) and scaled down (1000x).

Broadcast benchmark. A single file is staged from NFS to intermediate storage. Then, a workflow stage produces a file in intermediate storage, which is consumed by 19 tasks running in parallel. When this file is created, WOSS creates eagerly (i.e., while each block is written) the number of replicas as specified by the replication tag. When the nodes process the input file, they randomly select a replica to read from (giving preference to local blocks if available), avoiding a scenario where a storage node becomes a bottleneck. Each task produces an output file that is finally staged-out to NFS.

Reduce benchmark. 19 files are staged from NFS to intermediate storage, 19 processes run in parallel, one per machine, each consuming one input file and producing one file tagged with '*collocation*'. These files are consumed by a single workflow stage that writes a single file as output which is, then, staged out to NFS. WOSS stores staged-in files locally and prioritizes storing the files tagged with '*collocation*' on a single node, exposes data location, to enable the benchmark script to execute the reduce task on this node, avoiding the overhead of moving data.

Scatter benchmark. An input file is staged-in to the intermediate storage from NFS. The first stage has one task that reads the input file and produces a scatter-file on intermediate storage. In the second stage, 19 processes run in parallel on different machines. Each process reads a disjoint region of the scatter-file and produces an output file. A tag specifies the block size to match the size of the application reading region (i.e., the region of the file that will be read by a process). WOSS exposes fine-grained block location information to enable scheduling the processes on the nodes that hold the block. Finally, the stage-out phase, the 19 output files are copied to NFS.

Statistical Analysis. We report the average execution time and standard deviation (in error bars) of at least 12 runs (a large enough sample to guarantee a 95% confidence level for the average). When we report speedup of the workflow-aware solution over alternatives, we base this speedup on the average execution time and we report the p-value [39] for 95% confidence value.

Results. Figure 6, Figure 7, Figure 8 and Figure 9 present the average benchmark runtime and standard deviation for five different intermediate storage systems setups. (1) NFS; (2) two setups for DSS - labeled 'DSS-RAM' or 'DSS-DISK' depending on whether the storage nodes are backed by RAM-disk or spinning disks; and (3) two setups for WOSS, labeled 'WOSS-RAM' or 'WOSS-DISK'). A sixth configuration for a local file system based on RAM-disk in the pipeline benchmark only represents the best possible performance.

Overall, a WOSS-* system exhibits higher performance than the corresponding DSS-* system, which had better performance than NFS. This shows that the overheads brought by tagging, reading tags, and handling optimizations are paid-off by the performance improvements. Another advantage of WOSS is lower variance since it depends less on the network. Finally, while for small and medium workloads RAM and Disk configurations achieve comparable performance as files fit in the machine cache, for large files RAM-disk-based configurations perform faster and with less variability than their spinning disks counter-parts.

Locality in the pipeline scenario (Figure 6) was the optimization that provided the best improvements. In this case, WOSS is 10x faster than NFS, and 2x faster than DSS. Note that, although WOSS adds overhead of metadata operations to handle optimizations and has additional context switches and memory copies introduced by FUSE user-level file system, WOSS provides performance comparable to *local* (the best possible scenario). In fact, t-tests show that the performance of WOSS is to the local storage (less than 3% on average, p-value 0. 035).

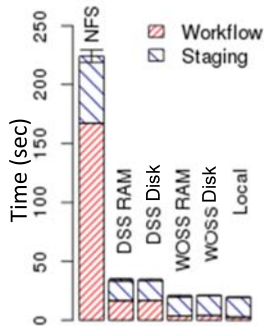

Figure 6. Average time for the pipeline benchmark.

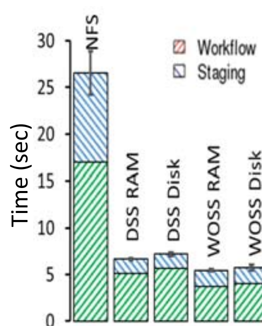

Figure 7 - Average time for the broadcast benchmark.

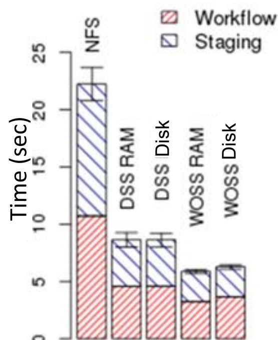

Figure 8 - Average time for the reduce benchmark

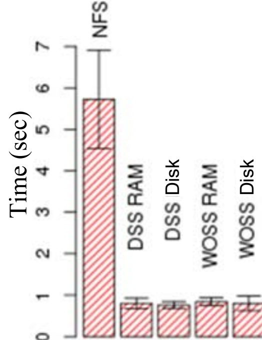

Figure 9 - Average time for stage 2 of scatter.

The broadcast benchmark (Figure 7) presents a more interesting case: Tagging for replication provides a finer tuning (number of replicas) for optimization than the other techniques that rely just on turning on/off (e.g., locality, and collocation). The number of replicas enables trading off the replication overhead with higher file read throughput. Figure 7 presents the performance for this benchmark when reaching the best performance (8 replicas). This result matches the expectation of the potential benefits of WOSS approach. For more replicas than optimal, the overhead of replication is higher than the gains. A simple heuristic (e.g., creating one replica for every 2-3 readers) can guide this optimization and provide most of the performance gains.

Staging and file creation for scatter benchmark takes a significant amount of the benchmark time (70-90%), and thus, for clarity of the presentation, Figure 9 focuses only on the workflow stage that is affected by the optimization. Following the same trend of pipeline benchmark, scatter is 6 times faster than NFS. WOSS, however, does not bring significant gains over DSS, performing similarly (with t-test giving p-values greater than 0.05). For the reduce benchmark, WOSS achieves 4.5x speedup compared to NFS and 1.5x compared to DSS. Due to the small file sizes, RAM and Disk configurations achieve comparable performance as files fit in the machine cache.

In addition to using the workload presented in Figure 5, we also executed the benchmarks with data sizes scaled up (10x) and scaled down (1000x). The larger workload had results similar to the ones presented in this section. The smaller one did not show significant difference among the storage systems (less than 10%, in order of milliseconds) with DSS performing faster than WOSS in some cases since the overhead of adding tags and handling optimizations did not pay off for such small files. Finally, as expected, for the larger files RAM-disk-based configurations also perform faster and with less variability than their spinning disks counter-parts.

Network Overhead. Table 4 shows the total network traffic generated by each of the benchmarks under the medium workload. WOSS brings significant saving in network overhead. While WOSS almost eliminates the network overhead for the pipeline benchmark, it saves 87% and 60% of the network traffic generated by the reduce and scatter patterns respectively. Surprisingly, WOSS brings network savings (7% reduction) even in the broadcast pattern which creates 8 replicas of the intermediary file. The replication extra network overhead is paid for by higher local access at the nodes storing the file. The small and large workloads achieve similar results.

Table 4. Generated network trafic (GB)

Benchmark	NFS	DSS	WOSS
Pipeline	18.60	17.60	1.86
Broadcast	2.87	2.03	1.91
Reduce	1.13	1.07	0.36
Scatter	0.60	0.57	0.27

Energy Consumption. We evaluated the energy saving the WOSS prototype brings. Since power meters are not widely available especially at scale, we used 8 nodes from Grid5000 'Lyon' cluster. Each node has two 2.3GHz Intel Xeon E5-2630 CPUs (each with six cores), 32GB memory and 10 Gbps NIC. The cluster nodes are instrumented with energy measurement module that is able to report the power consumed per node per-second granularity. Our evaluation shows that WOSS brings significant energy savings: between 20% and 50% compared to the DSS system, depending on the access pattern.

Summary of synthetic benchmark evaluation. Overall, WOSS exhibits higher performance than the corresponding DSS system, which has better performance than NFS. This shows that the overheads brought by tagging, reading the tags, and handling optimizations are paid-off by the performance improvements. Another two advantages of WOSS are lower network traffic and, as a result, lower performance variance since it depends less on the network.

WOSS gains vary for different benchmarks: 10x over NFS and 2x over DSS for pipeline benchmark, 4.5x over NFS and 1.2x over DSS for the reduce benchmark, 1.5x over NFS and 1.09x over DSS for the broadcast benchmark, and 6x over NFS for scatter. WOSS also brings significant savings in the amount of data transferred, almost eliminating network overhead for the pipeline benchmark, and saving 60%-87% of the network traffic generated by the reduce and scatter patterns respectively. Finally WOSS leads to lower performance variability.

Using the pipeline benchmark at scale (100 nodes) to compare with other systems. The benchmark runs multiple pipelines in parallel. Each pipeline stages-in a common input file from the shared backend (i.e., the NFS server), goes through three processing stages using the intermediate storage, then the final output is staged out back to backend. The script tags the output files produced by a pipeline stage with a *'local'* tag to inform the storage to attempt storing the output files on the node where the task runs. WOSS exposes the file location so that the benchmark can launch the next stage on the machine that already stores the file. The benchmark is presented in Figure 5.

We ran the pipeline benchmark at scale on our Grid 5000 testbed, TB100 where 99 pipelines run in parallel. We compare with Ceph and GlusterFS deployed as intermediate storage systems in addition to comparing with NFS, DSS, and node-local storage as in smaller scale experiments (Figure 10). Since Grid 5000 does not allow deployments of these systems on RAM-disks, we report numbers of Ceph and Gluster on spinning disk deployments only.

Figure 10 shows the boxplots for pipeline execution times on the different storage systems. Average execution time for NFS is 138 seconds, several times slower than the other systems, and omitted from the plot to reduce clutter. Ceph and Gluster perform faster than NFS, but slower than DSS and exhibit higher variability. At this scale, the gains brought by WOSS are more relevant than in the smaller-scale deployment resulting in a performance 6x higher than DSS (instead of only 2x on the smaller deployment). It is also possible to notice the effects of the scale on the storage system as WOSS is 2x slower than the optimal performance (estimated by running on a node-local file system) rather than similar as in the small deployment.

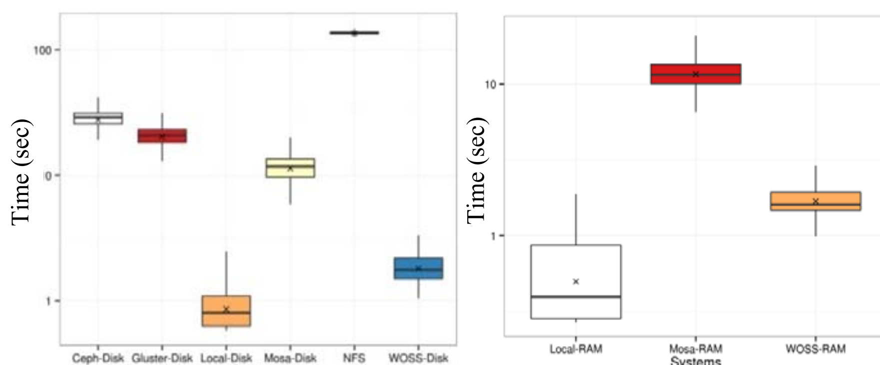

Figure 10 – *left* plot: Large-scale (100-nodes) pipeline benchmark performance for WOSS, DSS, Ceph and GlusterFS deployed over node-local spinning disk. *right* plot presents the performance for WOSS, DSS (Mosa), and local RAM-disk, when using only RAM-disks. Boxplots present average, median, 1st and 3rd quartile the execution time over 500 pipeline executions. *Note the logarithmic y-axes on both plots and the different scales.*

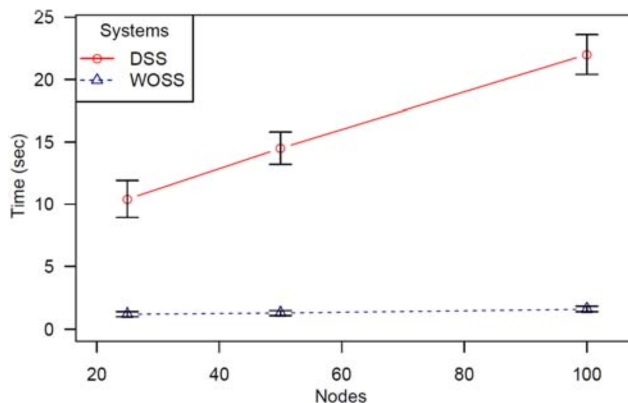

Figure 11 – Average execution time for pipeline benchmark in a weak scaling scenario (25, 50, and 100 nodes).

Scaling pipeline benchmark. One common challenging point for distributed storage systems is the potential bottleneck of having a, at least logically, centralized metadata manager. We argue that the cross-layer approach does not add significant overhead, and demonstrate through experiments that the system scales, despite the extra metadata operations to handle cross layer communication. To this end, we perform a weak scaling experiment based on the pipeline benchmark with 25, 50, and 100 pipelines each using one node. The data size is described in Figure 5 (left). As in the previous scenarios, the benchmark runs multiple pipelines in parallel. Each pipeline stages-in a common input file from the shared backend, goes through processing stages using the intermediate storage, and the final output is staged out to the backend. If WOSS is used, then the script tags the intermediate files as *'local'*. Figure 11 shows the performance for both systems. WOSS has just a 33% increase in time for the 4x increase in the system scale (25 to 100 nodes), while DSS exhibits a much worse runtime increase (120%). The savings that WOSS brings to the platform allow a smaller increase in the execution time, closer to a constant execution time as one would expect from a perfectly scalable system is a weak scaling scenario.

4.2 Simple Real Applications: BLAST, modFTDock

We use two relatively simple real-world applications and testbed TB20:

- *modFTDock* [40] a protein docking application. The workflow has three stages with different data flow patterns (Figure 12): *dock* (broadcast pattern), *merge* (reduce), and *score* (pipeline) stages.
- *BLAST* [41] a DNA search tool. Each node receives a set of DNA sequences as input (a file for each node) and all nodes search the same database file (i.e., BLAST uses the broadcast pattern to replicate the database) (Figure 14).

Integration with workflow runtimes. For *modFTDock* we use *Swift* to drive the workflow: As *modFTDock* combines broadcast, reduce and pipeline patterns, *Swift* tags the database to be *replicated* (broadcast), the output of every *dock* stages is *collocated* on a single storage node that executes the *merge* stage (reduce). The merge output is tagged to be placed on local storage node in order to execute the *score* stage on the same machine (pipeline). For *BLAST* we use shell scripting: the script tags the database file to a specific replication level and the input file as '*local*'.

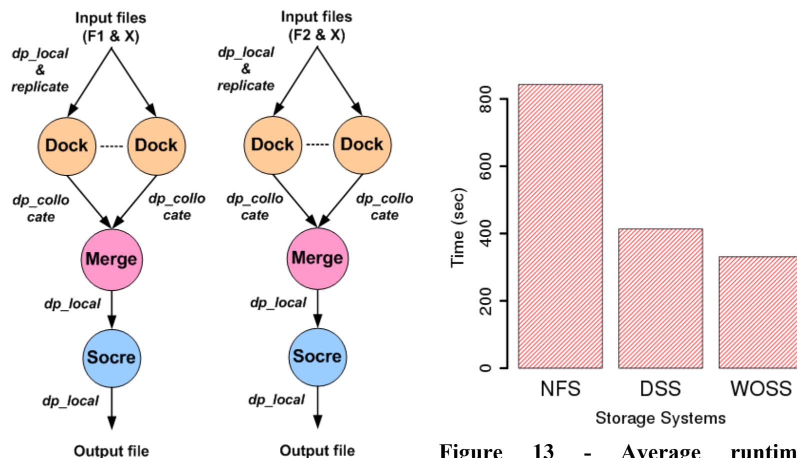

Figure 12 - *modFTDoc* worflow. Labels on arrows represent the tags used

Figure 13 - Average runtime for *modFTDock* on cluster. We run 9 pipelines in parralel using 18 nodes (average over 5 runs).

***modFTDock* experiments.** 9 dock streams progress in parallel and process the input files (100-200KB) and a database (100-200KB). The storage nodes are mounted on RAM-disks. Figure 13 presents the total execution time for the entire workflow including stage-in and stage-out times for DSS and WOSS. WOSS optimizations enable a faster execution: *modFTDock/Swift* is 20% faster when running on WOSS than on DSS, and more than 2x faster than on NFS.

BLAST experiments. 19 processes launch 38 DNA queries in the database independently and write results to backend storage. We report results from experiments that use a 1.7GB database. Output file sizes are 29 to 604KB. Table 5 presents the breakdown for the BLAST workflow runtime. Our approach offers performance gains compared to NFS (up to 40% better performance) and compared to DSS (up to 15% better performance).

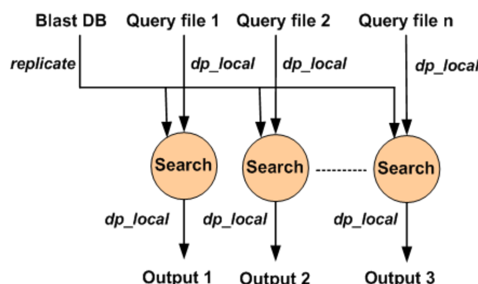

Figure 14. *BLAST* workflow. All nodes search *BLAST* database (1.8GB) in parallel for different queries. Labels on arrows represent the tags used to hint the data usage patterns.

Table 5. Average BLAST execution time (in seconds) for NFS, DSS and various replication levels controlled in WOSS.

			WOSS (replication factor)			
	NFS	**DSS**	**2**	**4**	**8**	**16**
Stage-in	49	17	19	29	36	55
90% tasks	264	185	164	155	151	145
All tasks finished	269	207	173	165	162	164
Stage-out	1	1.2	1.3	1.2	1.1	1.1
Total	**320**	**226**	**193**	**191**	**200**	**221**

4.3 A Complex Workflow: Montage

Montage [42] is a complex astronomy workflow composed of 10 different stages (Figure 15) with varying characteristics (Table 6). The workflow uses the 'reduce' pattern in two stages and the 'pipeline' patterns in 4 stages (as the labels indicating the tagging we have used and present on the arrows in Figure 15 indicate) thus offering an opportunity for cross-layer performance optimizations.

Workflow characteristics. The I/O communication intensity between workflow stages is highly variable (presented in Table 6 for the workload we use). The workflow uses *pyFlow*. Overall, the workflow generates over 12,000 files with sizes from 1KB to over 3GB and more than 30 GB of data are read/written from/to storage.

Systems tested. We executed Montage workflow over four configurations: Ceph, GlusterFS, DSS and WOSS deployed on testbed TB100 with node-local spinning disks. GlusterFS trials are not reported because the application failed to finish in reasonable time.

Results. Table 7 highlights that WOSS offers 25% better performance compared with DSS and 38% compared with Ceph. Table 7 also shows the average execution time per workflow stage. Overall, as expected, WOSS gains over DSS come from stages that move more data (e.g., mJpeg and mAdd). Note that some stages take longer on WOSS because of the overheads caused by adding the extended attributes to files (‘*tagging’*), getting the location of a file, and doing location-aware scheduling (e.g., mFitplane). On the other hand, the gains of DSS and WOSS over Ceph come mainly from stages that perform many operations in parallel (e.g., mDiff). We believe that mJpeg on Ceph takes advantage of caching from mAdd stage. We are investigating why mAdd is significantly slower than for both WOSS and DSS than in Ceph.

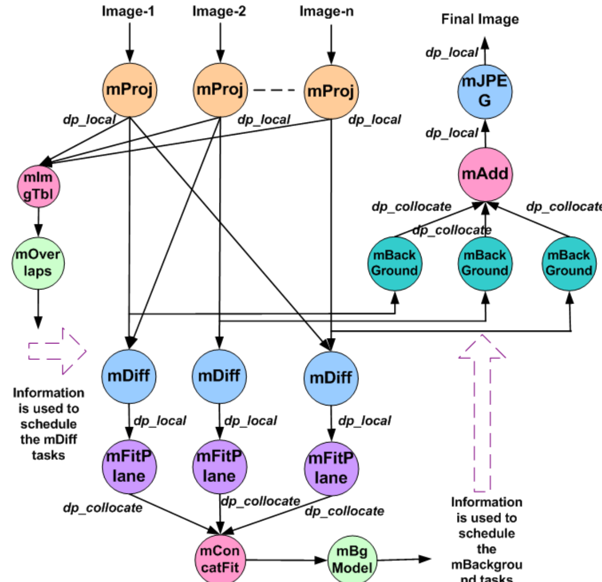

Figure 15. Montage workflow. The characteristics of each stage are described in Table 6. Labels on the arrows represent the tags used to indicate the data usage patterns.

4.3.1 Montage Scheduling Overhead

Montage workflow is composed of thousands of tasks potentially imposing a high overhead for scheduling and launching processes on different machines. The current WOSS prototype further adds non-negligible overheads for storing and retrieving extended attributes from the metadata manager for each of the workflow files. To understand the impact of these overheads we designed an experiment that highlights the scheduling overheads. To eliminating task execution time we have modified the workflow to execute `‘sleep 0’` instead of Montage tasks. Further, to eliminate the overhead of data movement we have modified the workload to use empty files. To keep the scheduling overhead equal to the scheduling overhead of the Montage workflow, we used the same workflow structure, same number of tasks, and the same number of files.

Overall, the execution time for `‘sleep 0’` scenario is 4% of DSS time. Table 7 shows the actual time to execute DSS and the time for ‘sleep 0’ for different stages. For the parallel stages that launch over 500 tasks (highlighted in italics in Table 7), the overhead to schedule and launch these tasks is up to 19 seconds (about 37ms per task). Note that for some stages (e.g., mDiff, mFitPlane) the scheduling overhead exposed by `‘sleep 0’` represents 45-67% of the actual stage execution time in DSS/WOSS, suggesting that there is a limited opportunity for additional gains in optimizing these stages.

For stages that should benefit from WOSS optimizations (marked with * in Table 7) the effectiveness of WOSS optimization varied based on the file sizes. mAdd and mJpeg, which process large amounts of data, do benefit from WOSS optimizations as it speeds up their writes and improves mJpeg reads. The extra overhead is not paid off for some of the other stages as the gains are smaller or comparable to the overhead of storing/retrieving file extended attributes to/from the metadata manager.

Table 6. Characteristics of Montage workflow stages. Optimized column shows the stages that should benefit from the tagging.

Stage	Data	#files	File size	Optimized
stageIn	1.9 GB	957	1.7 - 2.1 MB	
mProject	8 GB	1910	3.3 - 4.2 MB	Yes
mOverlaps	336 KB	1		
mDiff	2.6 GB	564	0.1 - 3 MB	Yes
mFitPlane	5MB	1420	4.0 KB	Yes
mConcatFit	150 KB	1		Yes
mBgModel	20 KB	1		
mBackground	8 GB	1913	3.3 - 4.2 MB	Yes
mAdd	5.9 GB	2	165MB	Yes
mJPEG	46 MB	1		Yes
stageOut	3 GB	3	46MB-3GB	Yes

Table 7. Average time for different storage systems and the time an evaluation of scheduling overheads (i.e., the time to execute the same task dependency graph with sleep 0 instead of the actual application task) for different stages of Montage (in seconds). The stages with massive parallelism are highlighted in italics. Values represent averages over 10 runs. Stages marked with * are scheduled according to the files'tags. Staging is omitted.

Stage	Ceph	DSS	WOSS	Sleep 0
*mProject **	*62.71*	*16.91*	*16.74*	*4.93*
mOverlaps	154.71	52.71	54.83	0.28
*mDiff**	*184.48*	*24.32*	*28.52*	*16.37*
*mFitPlane**	*58.94*	*42.38:*	*43.87*	*19.64*
mConcatFit*	39.43	41.51	43.68	0.14
mBgModel	31.10	31.30	31.33	0.14
*mBackground**	*63.33*	*15.59*	*18.78*	*8.48*
mAdd*	268.72	446.96	350.32	0.15
mJpeg*	116.52	264.76	119.89	0.16
TOTALS	***979.94***	***894.06***	***707.96***	***50.29***

4.3.2 Exploring WOSS Overheads/Gains

In the current implementation, to enable cross-layer optimization a set of operations are needed, including: forking a process to add a file extended attributes (labeled *fork* in Table 8), adding the extended attributes to files ('*tagging*'), getting the location of a file ('*get location*'), and doing location-aware scheduling ('*location-aware scheduling*'). All these steps add additional overhead and only after all three steps have been done can the benefits be reaped.

To guide future optimizations we run the same Montage workload as in the previous section yet configured to expose the overhead of each of these steps. For instance, to expose only the overhead of tagging, we tag the files, in all the benchmarks in Table 8 except WOSS, with a random tag that will add the overhead without triggering any optimization.

Table 8 shows the average execution time of the Montage workflow. The results suggest that steps described above add significant overhead (up to 7%). A closer look reveals that the tagging operation is the main contributor to the overhead. The main reason is twofold: first, every tagging operation incurs a roundtrip to the manager; second, the current manager implementation serializes all 'set-attribute' calls, this adds significant delay considering that Montage workflow produces and tags over 660 files in every run.

Our evaluation highlights that optimizing the 'set-attribute' operation (by eliminating the fork(), caching, and increasing the manager implementation parallelism) can bring significant additional (up to 7%) performance gains. We note that the use of fork was an implementation shortcut due to the limited abilities of the run time scheduler we use (that added 2% overhead).

Table 8. WOSS microbenchmark.

Experiment setup	Total time (s)
DSS	66.2
DSS + fork	67.1
DSS + fork + tagging	69.5
DSS + fork + tagging + get location	70
DSS + fork + tagging + get location + location-aware scheduling (on useless tags)	70.7
WOSS (all of the above with useful tags)	61.9

5 Discussion and Summary

Cross-layer optimizations bypass a restricted, 'hourglass', interface between system layers. A classic example is the TCP/IP stack: in the original design, the transport layer assumes that a lost packet is an indicator of congestion and backs-off. This assumption is violated in wireless environments and leads to degraded performance. To deal with this situation, a number of mechanisms expose the lower layers' state and channel capability such that the upper layer can infer the cause of packet loss and react appropriately.

Storage systems can be viewed through the same lens: the traditional (and after decades of use, convenient) POSIX file system API performs the role of the 'hourglass' neck. In the last two decades a number of systems proposed specialized APIs for passing applications hints to inform storage system optimizations. To date no widely used system adopts these approaches, as requiring changes to the standard API hinders adoption. This paper proposes using *standard extended* attributes in this role. We argue that this is a flexible, backward compatible, mechanism for communication between the storage and applications that unlocks an incremental adoption path for cross-layer optimizations in storage systems. We demonstrate this approach in context of workflow execution systems. WOSS exploits application hints to provide per-file optimized operations, and exposes data location to enable location-aware scheduling. The simple policies/hints we explore unlock sizeable performance benefits, suggesting that further work could yield bigger gains.

Design guidelines. Two design lessons can be borrowed from the design of the network stack: First, both applications and the storage should *consider metadata as hints rather than hard directives.* That is, depending on specific implementation and available system resources directives expressed through custom metadata might or might not be followed. Second, to foster adoption, *adding support for cross-layer optimizations should not (or minimally) impact the efficiency of applications or storage system not*

using them (otherwise these mechanisms are less likely to be adopted in practice as with some of the solutions that did not gain traction in the networking space [43]).

We put forward two additional design guidelines: First, *the cross-layer communication and the optimizations enabled should not break the separation of concerns between layer*s. A key reason for layered designs is reducing system complexity by separating concerns at different layers. Therefore, it is necessary to devise mechanisms that limit the interference one layer may cause on others even though, as we argue, there are benefits in allowing information to cross between layers. Second, the *distinction between mechanism and policy should be preserved*. The various policies associated with the metadata should be kept independent from the tagging mechanism itself.

Cross-layer optimizations in storage systems. Apart from the use cases discussed before a number of other optimizations are possible. We briefly list them here:

Cross-layer optimizations enabled by top-down information passing. Applications may convey hints about their: QoS requirements, versioning, or consistency requirements (e.g., to relax in certain conditions the consistency requirements for higher performance).

Cross-layer optimizations enabled by bottom-up information passing. In addition to enabling location aware scheduling, storage-level information could be useful when making application level-decisions as well. For instance, exposing file layout can enable efficient parallel read operations [44], exposing device specific performance characteristics can enable optimizing database operations [45], or matching the application access pattern to the disk characteristics [7], enable energy optimizations by exposing which nodes contain well replicated blocks and can be shut down, or in wide area setup, exposing file location and the network and storage bandwidth [46],

Determining the data access patterns (and thus the tags) is crucial to unlock the cross layer optimization opportunities we explore. As most applications use thousands of files and contain more than one pattern, hand-coding crafting the workflows, the approach we have used in our experiments, is not practical. The most direct approach to this issue is to build this functionality within the workflow runtime engine. The 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 then be provided to the underling storage system to optimize its operations based on these hints.

WOSS Overheads. To enable cross-layer optimization a set of operations are needed, including: forking a process to add a file extended attributes, adding the extended attributes to files (‘*tagging*’), getting the location of a file, and doing location-aware scheduling. All these steps add overhead. To guide future optimizations, we ran a small Montage workload on our cluster yet configured to expose the overhead. The results suggest that steps described above add significant overhead (up to 7%). The main reasons are twofold: First, every tagging operation incurs a roundtrip to the manager. Second, the current manager implementation serializes all ‘set-attribute’ calls, this adds significant delay considering that it produces and tags over 660 files in every run.

Cross-layer communication may make the configuration harder. To achieve the highest possible performance the system should identify the most effective configuration for that pattern. While it is relatively easy to identify the effective optimization for most of the workflow access patterns, this decision is harder in the general case. By allowing the application to specify the storage system configuration on the fly, cross-layer communication enables a number of possible configurations that was not possible or easily enabled before. This complexity emerges from the possibility of tuning the system to achieve the highest possible performance for a given application. If the user is willing to pay this trade-off of tuning the system to improve performance, she/he can opt to use different tools to aid this process. For example, our team [47, 48] has been working on a solution to quickly explore the configurations and recommend one for a given application on a given platform.

Limitations. The proposed approach and design have two main limitations. First, the proposed *per-file* cross-layer optimization approach assumes that data of each file is stored separately from the other files, this limits the use of this approach in systems in which a single data block can be part of multiple files (e.g. content addressable storage, or copy-on-write storage system) as it is possible for separate files that share a block to have conflicting application hints. Second, our design allows extending the system with new optimizations; an unacceptable design for secure storage system as it adds significant vulnerabilities.

Summary. This paper proposes using custom metadata as a bidirectional communication channel between applications and the storage system. We argue that this solution unlocks an incremental adoption path for cross layer optimizations in storage systems. We demonstrate this approach in context of workflow execution systems. Our workflow optimized storage system (WOSS), exploits application hints to provide per-file optimized operations, further, our system exposes data location to enable location aware-scheduling. Our evaluation shows that WOSS brings tangible performance gains.

6 References

1. Koren, J., et al. *Searching and Navigating Petabyte Scale File Systems Based on Facets*. in *ACM Petascale Data Storage Workshop, in Supercomputing* 2007. Reno, NV.
2. Leung, A.W., et al. *Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems*. in *FAST*. 2009.
3. Carns, P.H., et al. *PVFS: A Parallel File System for Linux Clusters*. in *4th Annual Linux Showcase and Conference*. 2000. Atlanta, GA.
4. Schmuck, F. and R. Haskin. *GPFS: A Shared-Disk File System for Large Computing Clusters*. in *1st USENIX Conference on File and Storage Technologies (FAST'02)*. 2002.
5. Fedak, G., H. He, and F. Cappello. *BitDew: a programmable environment for large-scale data management and distribution*. in *International Conference on High Performance Networking and Computing (Supercomputing)*. 2008.

6. Arpaci-Dusseau, A.C., et al., *Semantically-smart disk systems: past, present, and future.* SIGMETRICS Performance Evaluation Review, 2006. **33**(4): p. 29-35.
7. Schindler, J., et al. *Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics*. in *Conference on File and Storage Technologies (FAST)*. 2002.
8. Sage Weil, S.A.B., Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn. *Ceph: A Scalable, High-Performance Distributed File System*. in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*. 2006.
9. *Trinity/NERSC-8 Use-Case Scenarios*. 2013 [cited 2014.
10. Wozniak, J. and M. Wilde. *Case studies in storage access by loosely coupled petascale applications*. in *Petascale Data Storage Workshop*. 2009.
11. Shibata, T., 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.
12. Bharathi, S., et al., *Characterization of Scientific Workflows*, in *Workshop on Workflows in Support of Large-Scale Science*. 2008.
13. Yildiz, U., A. Guabtni, and A.H.H. Ngu, *Towards scientific workflow patterns*, in *Workshop on Workflows in Support of Large-Scale Science*. 2009.
14. Raicu, I., 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.
15. Foster, I., et al., *Swift: A language for distributed parallel scripting* Journal of Parallel Computing, 2011.
16. Bent, J., 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.
17. Ghemawat, S., H. Gobioff, and S.-T. Leung. *The Google File System*. in *19th ACM Symposium on Operating Systems Principles*. 2003. Lake George, NY.
18. Gupta, K., et al., *GPFS-SNC: An enterprise storage framework for virtual-machine clouds* IBM Journal of Research and Development 2011.
19. Rosenblum, M. and J.K. Ousterhout. *The Design and Implementation of a Log-Structured File System*. in *ACM Transactions on Computer Systems*. February 1992.
20. *ROMIO: A High-Performance, Portable MPI-IO Implementation*. 2013; Available from: www.mcs.anl.gov/romio.
21. Lofstead, J.F., et al. *Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)*. in *Proceedings of the international workshop on Challenges of large applications in distributed environments (CLADE)*.
22. Mandagere, N., J. Diehl, and D. Du. *GreenStor: Application-Aided Energy-Efficient Storage*. in *IEEE Conference on Mass Storage Systems and Technologies (MSST)*. 2007.
23. Fujimoto, K., et al. *Power-aware Proactive Storage-tiering Management for High-speed Tiered-storage Systems*. in *Workshop on Sustainable Information Technology*. 2010.
24. Abd-El-Malek, M., et al. *Ursa minor: versatile cluster-based storage*. in *FAST* 2005.
25. Mesnier, M.P. and J.B. Akers, *Differentiated storage services.* ACM SIGOPS Operating Systems Review, 2011. **45**(1).
26. Patterson, R.H., et al. *Informed prefetching and caching*. in *ACM symposium on Operating Systems Principles (SOSP)*. 1995.
27. Ramya, P., et al., *Provisioning a Multi-tiered Data Staging Area for Extreme-Scale Machines*, in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*. IEEE Computer Society.
28. *IBM software defined storage*. [cited 2014; Available from: http://www-03.ibm.com/systems/storage/software-defined-storage/.
29. Raicu, I., et al. *Falkon: a Fast and Light-weight tasK executiON framework*. in *SuperComputing*. 2007.
30. Raicu, I., et al., *The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems*, in *International symposium on High Performance Distributed Computing (HPDC)*. 2009.
31. Zhang, Z., et al. *AME: An Anyscale Many-Task Computing Engine*. in *Workshop on Workflows in Support of Large-Scale Science*. 2011.
32. Wilde, M., et al., *Swift: A language for distributed parallel scripting.* Parallel Computing, 2011.
33. Henry, M.M., R.B. Ali, and S.V. Sudharshan, */scratch as a cache: rethinking HPC center scratch storage*, in *Proceedings of the 23rd international conference on Supercomputing*. 2009, ACM: Yorktown Heights, NY, USA.
34. Maheshwari, K., et al. *Evaluating Storage Systems for Scientific Data in the Cloud*. Workshop on Scientific Cloud Computing (ScienceCloud), 2014.
35. Thain, D., C. Moretti, and Jeffrey Hemmes, *Chirp: A Practical Global Filesystem for Cluster and Grid Computing.* Journal of Grid Computing, 2009. **7**(1): p. 51-72.
36. Thain, D. and M. Livny. *Parrot: Transparent User-Level Middleware for Data-Intensive Computing*. in *Workshop on Adaptive Grid Middleware*. 2003. New Orleans, Louisiana.
37. Bolze, R., et al., *Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed.* International Journal of High Performance Computing Applications, 2006. **20**(4): p. 481-494.
38. Yang, H., L.B. Costa, and M. Ripean. *Energy Prediction for I/O Intensive Workflow Applications*. in *ACM Workshop on Many-Task Computing on Grids and Supercomputers*. 2014. New Orleans, LA.
39. Trivedi, K.S., *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. 2 ed. 2001: Wiley-Interscience.
40. *modFTDock*. [cited 2012; Available from: http://www.mybiosoftware.com/3d-molecular-model/922.
41. Altschul, S.F., et al., *Basic Local Alignment Search Tool.* Molecular Biology, 1990. **215**: p. 403–410.
42. Laity, A.C., et al. *Montage: An Astronomical Image Mosaic Service for the NVO*. in *Proceedings of Astronomical Data Analysis Software and Systems (ADASS)*. 2004.

43. Fonseca, R., et al., *IP Options are not an option*, in *Technical Report No. UCB/EECS-2005-24*. 2005.
44. Thakur, R., W. Gropp, and E. Lusk, *An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces*, in *Proceedings of The 6th Symposium on the Frontiers of Massively Parallel Computation*. October 1996.
45. Schindler, J., A. Ailamaki, and G.R. Ganger. *Lachesis: Robust Database Storage Management Based on Device-specific Performance Characteristics*. in *VLDB* 2003.
46. Park, S.-M. and M. Humphrey, *Data Throttling for Data-Intensive Workflows*, in *International Parallel and Distributed Processing Symposium (IPDPS)*. 2008: Miami, FL.
47. Costa, L.B., et al. *Supporting Storage Configuration for I/O Intensive Workflows*. in *ACM International Conference on Supercomputing (ICS)*. 2014.
48. Costa, L.B., et al., *Support for Provisioning and Configuration Decisions for Data Intensive Workflows* IEEE Transactions on Parallel and Distributed Systems, 2015.