

TOWARDS A HIGH-PERFORMANCE SCALABLE STORAGE SYSTEM FOR WORKFLOW APPLICATIONS

by

Emalayan Vairavanathan

BSc.Eng (Hons), University of Moratuwa, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 2012

© Emalayan Vairavanathan, 2012

Abstract

This thesis is motivated by the fact that there is an urgent need to run scientific many-task workflow applications efficiently and easily on large-scale machines. These applications run at large scale on supercomputers and perform large amount of storage I/O. The storage system is identified as the main bottleneck on large-scale computers for many-task workflow applications. The goal of this thesis is to identify the opportunities and recommend solutions to improve the performance of many-task workflow applications.

To achieve the above goal this thesis proposes a two-step solution. As the first step, this thesis recommends and designs an intermediate storage system which aggregates the resources available on compute nodes (local disk, SSDs, memory and network) and provides a minimal POSIX API required by workflow applications. An intermediate storage system facilitates a high performance scratch space for workflow applications and allows the applications to scale transparently compare to a regular shared storage systems. As the second step, this thesis performs a limit study on workflow-aware storage system: an intermediate storage that is tuned depending on I/O characteristics of a workflow application.

Evaluation with synthetic and real workflow applications highlights the significant performance gain attainable by an intermediate storage system and a workflow-aware storage system. The evaluation shows that an intermediate storage can bring up to 2x performance gain compared to a central storage system. Further a workflow-aware storage system can bring up to 3x performance gain compared to a vanilla distributed storage system that is unaware of the possible file-level optimizations. The findings of this research prove that an intermediate storage system with minimal POSIX API is a promising direction to provide a high-performance scalable storage system for workflow applications. The findings also strongly advocate and provide design recommendations for a workflow-aware storage system to achieve better performance gain.

Table of Contents

Abstract.....	ii
Table of Contents	iii
List of Tables	v
List of Figures.....	vi
Acknowledgements	viii
Dedication	ix
1. Introduction.....	1
1.1 Motivation.....	2
1.2 The Problem.....	4
1.3 The Opportunities	5
1.4 Proposed Solution	6
1.5 Methodology	7
1.6 Contributions.....	8
1.7 Research Publications	9
1.8 Thesis Structure.....	10
2. Background and Related Work.....	12
2.1 Background	12
2.1.1 Software Stack.....	12
2.1.2 Hardware Platform Example	14
2.2 Related Work – Storage Systems.....	16
2.2.1 Generic Distributed Storage Systems	17
2.2.2 Application-optimized Storage Systems.....	17
2.2.3 Highly Configurable Storage Systems.....	18
2.2.4 Co-designed Data Management Systems	18
3. An Intermediate Storage System: Alleviating the I/O Bottleneck.....	20
3.1 Intermediate Storage System Requirements	21
3.2 Intermediate Storage System Design and Implementation	22
3.3 Intermediate Storage Evaluation	24
3.3.1 Synthetic Benchmark.....	26

3.3.2	modFTDock	27
4.	A Case for Workflow-Aware Storage Systems	28
4.1	The Opportunities	28
4.2	Data Access Patterns in Workflow Applications	30
4.3	Determining the Data Access Patterns	34
5.	Workflow-aware storage system: An Opportunity Study	36
5.1	Hacks: Customizing MosaStore	36
5.2	Evaluation	38
5.2.1	Micro Benchmark: The Impact of Locality	39
5.2.2	Synthetic Benchmarks	41
5.2.2.1	Pipeline Pattern Evaluation	44
5.2.2.2	Broadcast Pattern Evaluation	46
5.2.2.3	Reduce Pattern Evaluation	48
5.2.2.4	Scatter Pattern Evaluation	50
5.2.3	Workflow Applications	51
5.2.3.1	Montage	52
5.2.3.2	modFTDock	55
6.	Conclusion	57
6.1	Future Work	58
	References	60
	Appendices	66
	Appendix A : MosaStore Functional and Design Specification	66

List of Tables

Table 1: Popular data access patterns generated by workflows. Circles represent computations. An outgoing arrow indicates 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. We use a notation similar to that used by Wozniak et al. [9].....	33
Table 2: File sizes for different workflow patterns.....	43
Table 3: The characteristics of each stage for the Montage workflow	53

List of Figures

Figure 1: Software stack to support workflow execution on large scale machines	13
Figure 2: Architecture of Blue Gene/P Supercomputer at Argonne National Laboratory	15
Figure 3: Intermediate storage system on supercomputers	20
Figure 4: MosaStore storage system architecture. The figure highlights the three high-level components the system access interface (SAI) sitting at the client; the manager that stores all system metadata, and the donor nodes that store data chunks.	23
Figure 5: Pipeline	26
Figure 6: Pipeline runtime on BG/P	26
Figure 7: modFTDock runtime on BG/P	27
Figure 8: A case for workflow aware storage system	29
Figure 9: I/O throughput when the storage node is backed by <i>spinning-disk</i> (left plot) and <i>RAMdisk</i> (right plot). For each plot there are two sets of columns presenting the write and, respectively, the read performance. Note that the axes use different scales in the two plots. Figures represent average throughput, and standard deviation in error bars, over 30 reads/writes.	41
Figure 10: Summary of synthetic benchmarks for pipeline, broadcast, reduce, and scatter 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.....	43
Figure 11. Pipeline pattern – small files. Average execution time (in seconds) for small file sizes. Error bars represent standard deviation for all stages of the workflow (the entire experiment time).	44
Figure 12. Pipeline pattern – medium files. Average execution time (in seconds) for medium-size file. Error bars represent standard deviations for the entire experiment.....	44
Figure 13. Pipeline pattern large files. Average execution time (in seconds) for large file sizes.....	45
Figure 14: Average execution time for broadcast synthetic benchmark with medium workload. All storage systems are deployed on spinning disks.....	47

Figure 15. Average execution time for broadcast synthetic benchmark with large workload. All storage systems are deployed on spinning disks.....	47
Figure 16. Breakdown of broadcast benchmark for the ‘medium’ workload.....	47
Figure 17: Reduce pattern. Average benchmark execution time (in seconds).	48
Figure 18. Scatter pattern medium files. Average execution time (in seconds) and standard deviation for the scatter stage of the benchmark (medium file sizes).....	51
Figure 19: Scatter pattern large files. Average execution time (in seconds) and standard deviation for the scatter stage of the benchmark (large file sizes).....	51
Figure 20: Montage workflow. The tags we use to indicate data usage patterns are presented in the figure. The characteristics of each stage are described in Table 3. Labels on arrow represent the data access patterns.	52
Figure 21: Montage workflow total execution time. Note that, to better highlight the differences, y-axis does not start at zero.	54
Figure 22: Montage workflow per-stage execution time.....	54
Figure 23: modFTDoc workflow. Labels on arrows represent the data access patterns.	55
Figure 24: modFTDoc workflow total execution time.	55

Acknowledgements

I would like to sincerely thank my advisor, Professor **Matei Ripeanu**, for his regular, sincere and insightful advice in the last three years. These past three years would not have been possible without his guidance and support. I could not have asked for a better supervisor, it has been a privilege to work with Dr. Ripeanu.

At this time I would like to extend my thanks as well to my colleagues: Samer Al-Kiswany, Mohammad H. Afrasiabi, Lauro B Costa, Abdullah Gharaibeh, Bo Fang, Hao Yang, Abmar Barros and Elizeu Santos-Neto for their technical discussions and feedback on my research. Thank you for enriching my learning and making graduate school an enjoyable and meaningful experience.

I am grateful to Zhao Zhang, Daniel S. Katz, Michael Wilde, Justin M Wozniak and Ketan Maheshwari - collaborators at Argonne National Laboratory and University of Chicago – for helping me to integrate and evaluate the system with applications and complex software stack in Blue Gene/P.

I wish to acknowledge my lecturers Prof. J. P. Karunadasa, Prof. Sisil P. Kumarawadu, Prof. J. Rohan Lucas and Dr. Thrishantha Nanayakkara at University of Moratuwa for their encouragement and igniting my interest in the engineering field during my undergraduate studies. Further I would like to thank Manoj Bandara, Umesh Wanigasekera, Akalanka Chandrasiri, Manoj Minor, Chinthaka Thilakarathna, Thirunathan.S and Abdul Carder at Millennium Information Technology for their support and insightful comments during my stay.

The advice, encouragement and friendship provided by all my friends at UBC have been greatly appreciated. And last but not the least; I am grateful to my family who has been my first and foremost source of support and inspiration. I could not have come this far without my parents, Vairavanathan and Sarojinie for their unconditional love, for kindling my passion to learn and giving me the opportunity to come to Canada to do my Master's.

Finally I would like to thank United States Department of Energy (DOE) for the financial support during my research.

To my family and my teachers

1. Introduction

Meta-applications that assemble complex processing workflows using existing applications as their building blocks are becoming increasingly popular. Examples include various scientific workflow applications (e.g., modFTdock [1], Montage [2], PTMap [3]) and even map-reduce applications [4]. Scientists in various fields like bio-chemistry, molecular-dynamics, geo-technology use these workflow applications to run massive simulations in order to solve critical problems. These applications run on supercomputers with many thousands of cores (e.g.: Argonne BG/P with 163,840 cores) and perform large amount of storage-I/O.

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 processes that communicate through intermediary files stored in a shared data-store 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], Makeflow [8]) are simplicity, direct support for legacy applications, and natural support for fault tolerance. First, a shared storage 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 the state of an application is persisted via files. A programmer can inspect the intermediate computation state at runtime and, if needed, can collect the files for performance profiling or offline debugging. Second, most of the legacy applications that form the individual workflow stages are coded to read their input and write their output through the well-known 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 performance is reduced due to the various bottlenecks in both hardware and software stacks on the workflow platforms [9][10].

The goal of this thesis is to identify the opportunities and recommend solutions to improve the performance of many-task workflow applications. To this end this thesis proposes and designs an intermediate storage system to increase the performance of workflow applications on large-scale machines. Further this thesis studies the feasibility of tuning the intermediate storage system depending on the application characteristics and proves that the intermediate storage system can be further optimized to match the high-level data access patterns of the workflow applications.

This chapter is organized as follows. Section §1.1 describes the motivation behind this thesis by highlighting the benefits and the importance of this research. Section §1.2 explains the current problems faced by the workflow applications. Section §1.3 highlights the available opportunities and briefly discusses possible approaches to harness these opportunities. Sections §1.4 and §1.5 present our solution and methodology respectively. Sections §1.6 and §1.7 summarize the contribution and the list of publications resulted from this thesis. Finally section §1.8 presents the structure of the thesis.

1.1 Motivation

The many-task workflow approach is becoming more popular since it provides high flexibility, portability, a natural way to express parallelism and rapid application development.

This thesis is motivated by the fact that there is an urgent need to run the scientific many-task workflow applications efficiently, reliably and easily on large-scale supercomputers. Taking the initiative to advance the many-task workflow applications' performance will provide multiple benefits. Firstly improving the application runtime will lead to better resource utilization and energy savings. Secondly, present supercomputers are specially optimized to provide the best performance for MPI-based applications and such initiatives will eliminate the need for special costly hardware-level optimizations to run many-task workflow applications. Thirdly, considerable portion of supercomputers' time is occupied in running the workflow applications. The domain scientists run numerous scientific workflow applications on these machines to solve useful and challenging problems. Fourthly, providing better scalability and high performance for many-task workflow applications will help the domain scientists to solve larger problems more accurately than the problems solved presently. From this point onwards whenever we say workflows we only refer the many-task computing workflows.

The following section summarizes three popular scientific workflow applications and their characteristics to emphasize above facts such as usefulness, efficiency, scale and amount of storage I/O.

- **ModFTDock:** Is a popular protein docking application used in new drug designs [1]. It processes a set of protein molecule's 3D spatial properties and finds high affinity molecules towards a particular protein. ModFTDock is also used to dock RNA to protein molecules. Protein docking applications such as ModFTDock and OOPS typically runs at scales ranging up to 65,536 of CPUs and perform massive amount of storage IO (read 3.2 PB and write 2 PB) [9].
- **Montage:** Montage workflow [2] is an astronomy application that takes many input images, processes and produces a single output. The workflow is composed of multiple

stages and has been used by scientists with parallel frameworks such as MPI, Pegasus [7], and Swift [6]. Montage typically runs on thousands of machines and applications of Montage includes but not limited to data analysis, quality assurance and scientific product generation [2].

- **BLAST** (The Basic Local Alignment Search Tool): Is a well adopted workflow application to solve sequence alignment problems such as protein structure prediction, pattern identification, phylogenetic analysis and etc. It uses a heuristic method and searches one or more nucleotide or protein sequences against a sequence database, and calculates similarities. Typical BLAST workflow application runs on thousands of machines and reads 3.5 PB of data and writes 150GB of data from / to the central storage system [9].

1.2 The Problem

Workflow applications can be viewed as a graph of distinct tasks which communicate through files stored on a central storage system. Each task takes an input file from the central storage system and processes it and produces an output file on the central storage system. *The central storage system has been identified as the main bottleneck for the scientific workflow applications on the prevalent super computers such as Blue Gene/P and Jaguar* [9][10]. Due to the limitations in both hardware and software, the storage performance drops severely when the workflow applications perform large number of storage IO operations.

First, on the hardware side, the IO bandwidth between compute nodes and storage nodes is limited hence the IO throughput drops linearly with the volume of IO operations. Second, POSIX file system abstraction has a strict semantic hence does not scale well at large scale and reduces the performance of workflow applications. Third, 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.

1.3 The Opportunities

The nature of workflow applications and the current supercomputing infrastructures used to run these applications provide three opportunities.

First, the compute nodes are underutilized and the compute node resources can be aggregated to produce a high performance shared storage space. The workflow based processing used by a large number of scientific applications [9], is generally composed of three main phases: stage-in input-data from central (and often external to the compute nodes cluster) storage to the compute nodes local storage, multiple computation stages that communicate through intermediate files, and stage-out the final results to the central storage. These three phases impose an intense workload on the central storage system. To reduce the load on the central storage applications can temporarily deploy and configure a shared intermediate storage to aggregate the storage resources available on the compute nodes (local disk, SSDs, memory) and use the storage system thus created as a high-performance scratch space to achieve better performance and resources utilization during runtime.

Second, strong POSIX storage abstraction adopted by central storage systems is not required due to the nature of the workflow applications. The costly locking protocol and the strict consistency guarantee provided by POSIX storage abstraction has been identified as the main scalability and performance bottleneck for parallel applications at the large scale [9]. In workflow applications, the workflow runtime has the entire knowledge of the workflow hence the file system consistency can be explicitly managed by the runtime and strong consistency provided by POSIX storage abstraction is not necessary. A storage system that covers the

minimal POSIX semantics that are required by workflow applications is enough to support a large number of legacy workflow applications and will be a promising solution to meet the scalability and the performance demands of the workflow applications at the large scale.

Third, workflow applications have regular data access patterns hence a storage system can be designed with special optimizations to speed-up these patterns [9][11]. Further an intelligent storage system / software stack can discover the data access patterns (either during application deployment time or runtime) of the workflow applications and harness these special optimizations depend on the patterns to provide better storage performance.

1.4 Proposed Solution

This work harnesses the opportunities described in section §1.3 and proposes a two-step solution to improve the performance of workflow applications.

As a first step this thesis proposes an intermediate storage system: A storage system that aggregate the storage resources available on the compute nodes (local disk, SSDs, memory) and provides the minimal POSIX API required by the workflow applications. At the beginning the workflow manager will stage-in the data from central storage system to intermediate storage system. Then the application does perform all the computations by reading from / writing to the intermediate storage system. At the end, the workflow manager will stage-out the final result back to the central storage system. The intermediate storage system will reduce the pressure on the central storage system as the amount of data generated during the workflow execution is much larger than the data staged-in or staged-out. Further it provides better scalability and performance for workflow applications compare to a regular storage system due to the minimal POSIX support.

As a second step the thesis investigates the viability of a workflow-aware storage system: that is, a storage system that is able to efficiently support the data access patterns generated by

workflows through optimizations at the file or directory level. Further, the storage system exposes data placement information so that the workflow runtime engine can make data-aware scheduling decisions. The storage system can identify the data access patterns either during application deployment time or application runtime to drive the data layout (e.g., co-placement, replication levels, local-chunk placement) of a workflow application [12], [13], [14]. *Storage system designers can incorporate the techniques that we study here in an intermediate storage system and can make it workflow-aware to provide an optimized storage performance for each application.*

1.5 Methodology

To build an intermediate storage system, we derive the requirements of an intermediate storage system by analyzing popular workflow applications. Then we design and implement the storage system. Finally we integrate the intermediate storage system with workflow runtime engine [6] and quantitatively evaluate the performance of synthetic and real workflow applications on Blue Gene/P supercomputer [15].

Then we systematically perform a limit study on workflow-aware storage system. We start from the previous characterization studies on workflow applications, extend the data access patterns by looking at the real workflow applications and identify the storage level optimizations to improve the regular data access patterns in the workflow applications. Then we quantitatively evaluate the impact of the optimizations on each data access pattern. We prove that a workflow-aware storage system can bring significant gains and also provide evidence to show that these techniques can be incorporated in an intermediate storage system with minimal effort [16].

1.6 Contributions

This thesis makes several contributions under two themes: first, an Intermediate Storage System; second, A Workflow-aware Storage System. The following paragraphs describe each contribution.

An Intermediate Storage System: A storage system that aggregates the resources of the computing nodes can provide temporary high-performance scratch space for the workflow applications. This thesis first, designs and implements an intermediate storage system. Second, it integrates the intermediate storage system with workflow runtime. Third, this study quantitatively evaluates the performance of intermediate storage system on a Blue Gene/P supercomputer [15] with synthetic and real workflow applications.

The design and implementation of the intermediate storage have been collaboratively performed with Samer Al-Kiswany and with minor help from a few other students at NetSysLab [17]. I contributed to the design, implementation and integration of the entire system. *Importantly I was the sole contributor for building the crucial components of an intermediate storage system for workflow applications on large scale machines. I designed and implemented mechanisms to support a set of POSIX APIs (random read and write), a part of metadata service, data placement policies, garbage-collection, chain and parallel replication, and client side caching of the intermediate storage system.* Section §3.2 highlights the significance of having these functionalities in an intermediate storage system designed to support workflow applications.

Even though my sole contribution is a complex effort (*30% of the entire code base which has more than 60,000 lines of code in total*), this document does not present the entire design of the system to keep the thesis concise. However an evaluation of the intermediate storage system

is presented in Chapter 3. For reference please refer the relevant sections of MosaStore technical design document attached in Appendix A .

Further it is important to note that building an intermediate storage system is essential to study the potential benefits of a workflow-aware storage system.

A Workflow-aware Storage System [16]: First, the thesis starts from previous workflow characterization studies, identifies new data access patterns by looking at some workflow applications and suggests corresponding storage-level optimizations to speed up each data flow pattern. Second it studies the viability of a storage system optimized for workflow applications: a storage system that can be optimized depending on the workflow application characteristics (i.e. based on data access pattern). It quantifies the potential performance benefits of a workflow-aware storage system. These suggestions can be incorporated in an intermediate storage system to further improve the performance of workflow applications. *These findings are published in CCGRID '12 [16] and also submitted for Elsevier FGCS Journal [18].*

In addition, these findings also motivated and informed the design of two research projects in the group: cross layer optimized storage system (Al-kiswany et al. [19]) and an automatically performance optimized storage system (Costa et al. [14]).

1.7 Research Publications

This work resulted in one refereed publication, one technical report, two conference submissions and one journal submission. These articles were written in collaboration with *Samer Al-Kiswany, Lauro Beltrão Costa, Hao Yang, Abmar Barros, Gillies Fedak, Zhao Zhang, Daniel S. Katz, Michael Wilde, and Matei Ripeanu*. The list of the articles is given below.

- *Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Beltrão Costa, Zhao Zhang, Daniel S. Katz, Michael Wilde, and Matei Ripeanu. A Workflow-Aware Storage System:*

An Opportunity Study. In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '12). Acceptance rate: 27% (83/302) - Nominated as one of the top 15 papers in the conference and invited for a special issue of Elsevier Future Generation Computer Systems Journal (impact factor 1.978) [16].

- ***Emalayan Vairavanathan**, Samer Al-Kiswany, Abmar Barros, Lauro Beltrão Costa, Hao Yang, Gilles Fedak, Zhao Zhang, Daniel S. Katz, Michael Wilde, and Matei Ripeanu. A case for Workflow-Aware Storage: An Opportunity Study using MosaStore. Submitted to special issue of Elsevier Future Generation Computing Systems Journal [18].*
- *Samer Al-Kiswany, **Emalayan Vairavanathan**, Lauro Beltrão Costa, Hao Yang and Matei Ripeanu. The Case for Cross-Layer Optimizations in Storage: A Workflow-Optimized Storage System. Submitted to FAST '13 [19].*
- *Lauro Beltrão Costa, Abmar Barros, **Emalayan Vairavanathan**, Samer Al-Kiswany and Matei Ripeanu. Predicting Intermediate Storage Performance for Workflow Applications. Submitted to CCGRID '13 [14].*
- *Samer Al-Kiswany, **Emalayan Vairavanathan**, Lauro Beltrão Costa, Hao Yang and Matei Ripeanu. MosaStore functional and design specification (Technical Report) [17].*

1.8 Thesis Structure

The rest of the thesis is organized as follows. In chapter 2, we present the software and hardware systems used by the workflow applications and explain the challenges faced by these applications. In chapter 2, we also provide a short summary of the related work on storage systems. In chapter 3, we propose an intermediate storage system, derive the requirements for the intermediate storage system in the context of workflow applications and also present the intermediate storage system evaluation. In chapter 4, we argue that there are still opportunities to

optimize the intermediate storage depending on the workflow applications' data access patterns and build a case for a workflow-aware storage system. In chapter 5, we study the opportunity of building a workflow-aware storage system using synthetic and real workflow applications. Finally we summarize our findings in chapter 6.

2. Background and Related Work

The chapter begins with a brief summary of a sample ecosystem that is used to support scientific workflow applications and the present challenges faced by these applications (§2.1). The chapter concludes with a high level summary of related work in storage systems (§2.2).

2.1 Background

Many-task workflows applications are composed of distinct executables with interdependencies and generally viewed as a graphs of dissimilar tasks that communicates though files stored in a shared storage system via POSIX API [20]. This style of workflows is easy to develop, debug, optimize and have implicit fault tolerance as the program state is persisted in a shared storage. Further structuring the application as many-task often provides a natural way to express parallelism and makes development easy compared to other approaches such as message passing [20]. The following sections present one example of the software and hardware stack used to support workflow applications on current large scale platforms.

2.1.1 Software Stack

Running workflow applications efficiently, reliably, and easily on large parallel computers is challenging. The workflow runtime can be roughly divided into two parts: a front-end that which evaluates a user workflow program and generate tasks (often a dataflow script written in a parallel scripting language), and a back-end which distributes the tasks to workers using a task dispatching service. Even though there are many ways to develop many-task workflows, the parallel scripting is a popular approach among scientists [20]. In a parallel scripting language, the workflow is often written as a script which assembles distinct interdependent tasks with each task reading and writing from a shared storage system. Swift [6], Dryad [21], Skywriting [22] , and CIEL [23] are few examples for such systems.

In this research we focus on Swift [6]. Swift is a scripting language designed for composing standalone executables into parallel applications that can be executed on compute nodes. Swift is widely used by domain scientists to write the workflow applications. Swift is responsible for the execution, composition, and coordination of the graph of tasks in a workflow application. Swift can automatically identify the task independencies and parallelize the programs to keep the resources utilization at high.

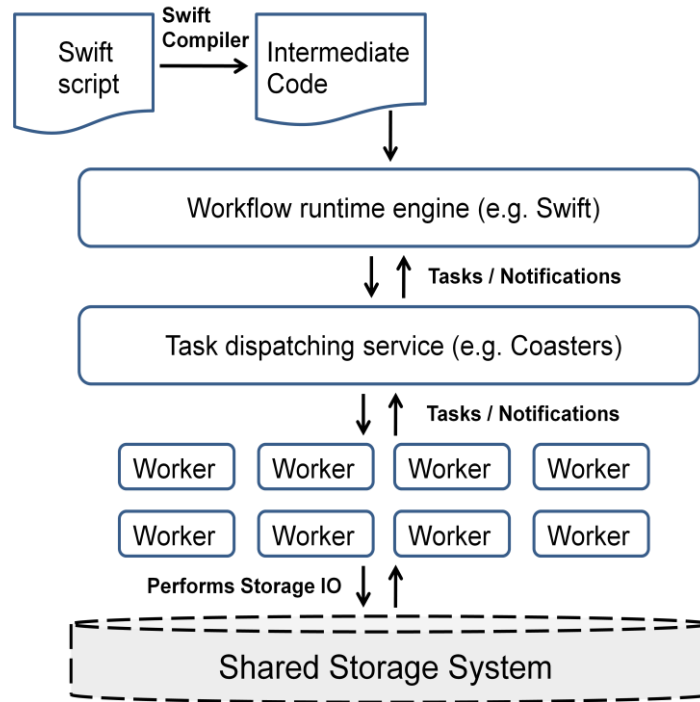


Figure 1: Software stack to support workflow execution on large scale machines

MapReduce [4] is another programming framework similar to Swift. Swift and MapReduce frameworks share a set of common design goals such as programmer productivity, implicit parallelism, transparent scalability, failure tolerance and load balancing. However comparing to MapReduce, Swift provides additional benefits such as portability, flexible data

model and intuitive programming model. Further Swift also supports multiple data access patterns whereas map-reduce only supports map and reduce data access patterns.

Figure 1 illustrates the software stack used by the workflow applications on large scale machines. A workflow application written in Swift is converted to intermediate code by the Swift compiler and then processed by the Swift workflow runtime engine [6]. The workflow runtime engine applies several optimizations such as automatic parallelization, adding heuristics to harness data locality and schedules the tasks on the compute nodes using a task dispatching service and the workers deployed on the compute nodes (e.g. Coasters [24]). The tasks read the input data from a shared storages system (generally the shared central storage system), process the data and write the output back to the shared storage system. Here it is important to notice the inter task communication is done through the files created on the shared storage system.

The shared storage system used in large scale machines generally supports a strong POSIX file system semantics and often incapable of handling the I/O demands of the scientific workflow application. For example GPFS starts to perform poorly when files are created under a same parent directory [10], [9] due to the strong consistency semantics and costly locking protocol.

2.1.2 Hardware Platform Example

Figure 2 illustrates the high-level architecture of BG/P super computers at Argonne National Laboratory (ANL) [15]. It has 160k compute cores, 640 I/O nodes (each with 4 cores) and 80 TB aggregated local memory and can provide 557TF aggregated peak performance [15], [25], [26]. Each compute node has 32 bit, 850 MHz 4 cores (IBM Power PC 450) and 2GB of memory.

The compute nodes are connected to two different networks called the ‘torus’ network and ‘tree’ network. In BG/P inter connect (or process) traffic uses the torus network and each compute node are connected to 6 neighbors via 6 torus links each have 6.8 Gb/s (3.4 Gb/s uni-

directional). I/O traffic uses the dedicated tree network and forwarded to designated IO nodes. Then the I/O nodes forward the I/O requests to the central storage system (e.g. GPFS / Lustre) often deployed on multiple storage nodes.

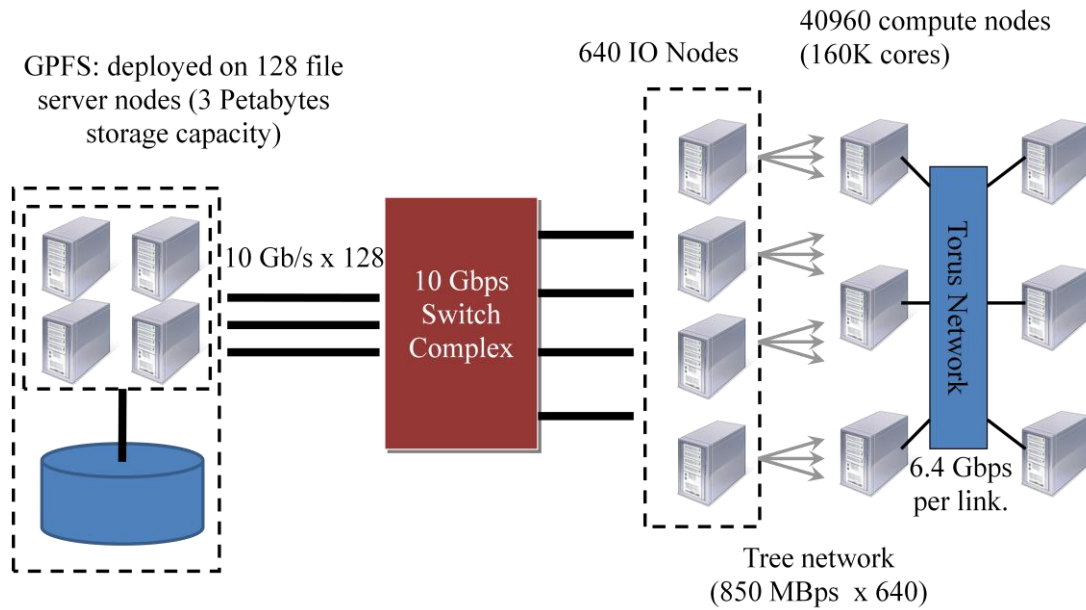


Figure 2: Architecture of Blue Gene/P Supercomputer at Argonne National Laboratory

Here we like to stress two important points.

- *First the architecture described above (incorporating compute nodes, I/O nodes and central storage via multiple high performance networks) is the defacto standard for large scale high performance computing systems.*
- *Second the central storage system becomes a bottleneck due to the limited bandwidth between the compute nodes and the storage nodes. For this machine a network with 1.28TB/s bandwidth connects the 160K computes nodes to the GPFS, resulting in only 1MB/s storage bandwidth per core.*

During the execution of workflow application a large number of tasks use the central storage system for inter-task communication and the central storage systems are incapable of handling the volume and frequency of the file system access generated by these applications. Hence time spent in I/O proportionally increases with the number of concurrent tasks and leads to low system performance. For example, Zhao et al. [27] characterize the Montage workflow execution time on BG/P super computers with 512 cores and the central GPFS storage system. The study shows that even at a very small scale 73.6% of total time is spent on performing I/O operations and waiting for I/O to complete due to I/O bottlenecks presents in the system. The actual processing time was 13.4% and the rest of the time (13.0%) was spent on other overheads such as scheduling.

2.2 Related Work – Storage Systems

The volume of past work on alleviating storage system bottlenecks is humbling. Distributed storage systems have been an active research topic for many years. This section limits itself to positioning a smaller number of projects that directly focus on alleviating the storage bottleneck for workflow applications. The rest of the section, first summarizes the work related to generic distributed storage systems for workflow applications (§2.2.1). Second, this section discusses the solutions optimized for a specific set of applications (§2.2.2). Third, this section summarizes the directions towards highly configurable storage systems (§2.2.3) – systems optimized for storage performance while preserving the shared storage system abstraction. Fourth, this section presents the work related to co-designed data management systems (§2.2.4) – this approach holds the promise of higher performance yet it leads to a more complex and less portable design by breaking the layering between the storage and the workflow runtime which offers a natural abstraction and separation of concerns.

2.2.1 Generic Distributed Storage Systems

Generic storage systems are designed with “*one size fits all*” philosophy to support all the applications. Examples for such storage systems include GPFS [28], Lustre [29] and Frangipani [30]. The design goal is to provide a transparent, reliable and secure storage abstraction to support most of the applications. Even though this approach has few advantages such as portability, security and ease of development, the main disadvantage is the limited performance. Storage abstraction provided by POSIX standard [31] is widely adopted in software systems. The POSIX storage abstraction was designed for single node file system and has severe scalability and performance bottlenecks when it is adopted by parallel file systems. The consistency semantics requires all the file system operation to be atomic; hence locking overheads exponentially increases at large scale with scientific workflow applications.

2.2.2 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 [32] optimizes for large datasets and append access, HDFS [33] which shares many design goals with GPFS-SNC [34], optimizes for immutable data sets, location-aware scheduling and rack-aware fault tolerance; the log-structured file system [35] optimizes for write intensive workloads, arguing that most reads are served by ever increasing memory caches and storage systems implementing the MPI-IO API optimize for parallel access operations. BAD-FS [36] optimizes for batch workloads, Amazon Dynamo [37] optimizes for intensive put/get operations and TidyFS [38] is designed to increase the performance of a set of Dryad [21] and DryadLINQ [39] applications and it supports high throughput, write once sequential I/O.

These storage systems and the many others that take a similar approach are optimized for one specific access pattern or set of applications and consequently are inefficient when different

data objects have different patterns, like in the case of workflows. Another known problem with application-optimized storage systems are non-standard APIs which makes the applications less portable between different systems.

2.2.3 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 [40] argues that storage systems should be specialized for the target application at deployment time. It aims to incorporate a broad set of optimization techniques, enable high configurability at deployment and/or run time, and support multiple applications through customized, per application deployment, all while still providing a standard POSIX API. In the same vein, Ursa Minor [41] 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. BFS [36] is an application optimized storage system yet it allows application instances to choose optimal caching, replication and consistency policies via a high level job scheduler. PPFS [42] is another configurable storage system which provides mechanisms to control the caching, pre-fetching, data layout and consistency semantics via a non standard API.

2.2.4 Co-designed Data Management Systems

A frequently adopted approach (Falkon [43][44], AME [45][27], Pegasus [7], GrADS [46], DAGMan [47]) for managing intermediate files in workflow runtime engines is to give up the shared storage system abstraction as well as the POSIX interface and redesign, from scratch, a minimal data store service coupled with the workflow runtime engine. The data store 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, schedule explicit data moves so that data is available on the local storage of a node before a task starts executing, or provide a global metadata service such that each compute node can check the availability of, and copy to local storage node, the intermediate input files before executing the task. 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 not 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.

3. An Intermediate Storage System: Alleviating the I/O Bottleneck

We propose a design for an intermediate storage system to alleviate the I/O bottlenecks faced by many-task workflows. An intermediate storage system aggregates the storage resources available on the compute nodes (local disk, SSDs, memory) and provides a high performance scratch space with minimal POSIX API (Figure 3).

An intermediate storage is deployed per application and has a limited life-time. A workflow runtime deploys and configures the intermediate storage on the compute nodes allocated for an application. Afterward the workflow runtime stages-in the data from the central storage system and starts the workflow application. The application performs all the computation on the intermediate storage and then the workflow runtime stages-out the final result to the central storage system. Finally the workflow runtime terminates the intermediate storage deployment and release the compute nodes.

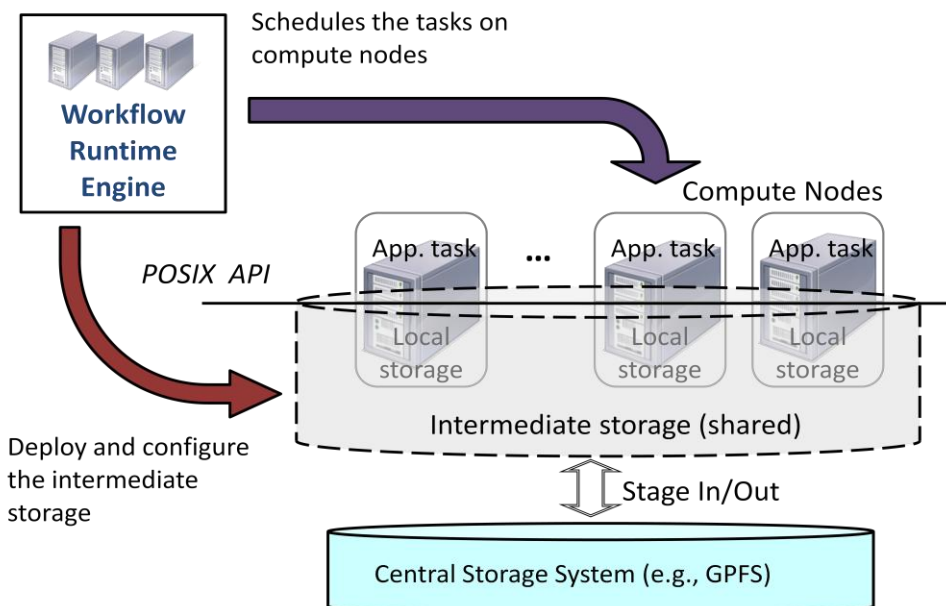


Figure 3: Intermediate storage system on supercomputers

Intermediate storage provides multiple benefits. First an intermediate storage system provides scalable and high performance storage as it can harness ample network bandwidth provided by the torus network. Second, it reduces the pressure on the central storage system as the amount of data generated during the workflow execution is much larger than the data staged-in or staged-out. Third, having a dedicated intermediate storage per application will reduce the interference from other users and also provide an opportunity to tune the storage system to achieve better performance.

3.1 Intermediate Storage System Requirements

There are multiple ways to design an intermediate storage system. For example Zhao et al. [27] give up the POSIX abstraction and uses a specialized interface for data management tasks. Providing a storage system with POSIX interface brings multiple benefits such as ability to integrate with legacy applications (without changing the applications), portability and rapid application development (POSIX is a well adopted standard).

An ideal intermediate storage system has the following general requirements to efficiently support various workflow applications.

- ***Easy to deploy:*** The storage system should be easy to deploy and mount by the workflow runtime during an application's initialization period (e.g., to support glide-in deployments [36] [48]). Further, ideally it should be transparently interposed between the application and the system central storage for automatic data pre-fetching or storing persistent data or results.
- ***Easy to integrate with applications:*** Majority of the workflow applications perform standard file system operations such as file creation and deletion, sequential / random reads and writes, directory creation and deletion. They do not use other complex file system operations related to permissions and users, file system locking and memory

mapping. A storage system that offers a partial POSIX-like API is adequate and can provide access to the aggregated storage space, without requiring changes to applications. Further strong consistency semantics is not required for an intermediate storage system as managing consistency can be moved to the workflow run-time. Offering additional, high-performance APIs (e.g., HDF5, NetCDF) might be desirable.

- ***Versatility and ability to configure:*** The storage system should provide several configuration knobs to support configurability for diverse applications. The system should be easy to configure and tune for a specific application workload and deployment environment. This includes ability to control local resource usage, in addition to controlling application-level storage system semantics, such as consistency and data reliability requirements.
- ***Efficiently harness allocated resources to offer high performance and scalability:*** The storage system should efficiently use the node-local storage and networking resources to provide high performance access to the stored data. To this end we choose the object storage approach for the intermediate storage system as it decouples data from metadata and provides the ability to scale data and metadata independently.

3.2 Intermediate Storage System Design and Implementation

We designed and developed an intermediate storage system by adding functionalities to MosaStore to make it capable of supporting workflow applications on supercomputing deployment. MosaStore (Figure 4) is an experimental shared storage system. MosaStore is designed to harness unused storage space from network-connected machines to build a high-performance, yet low-cost data store that can be easily configured for optimal performance in different environments. MosaStore aggregates distributed storage resources: storage space (based on spinning disks, SSDs, or memory) as well as the I/O throughput and the reliability of the

participating machines. Further, MosaStore adopts an object-based distributed storage system architecture, with three main components: a centralized metadata manager, the storage nodes (donor nodes), and the client's system access interface (SAI) which uses FUSE [49] kernel module to provide a POSIX file system interface.

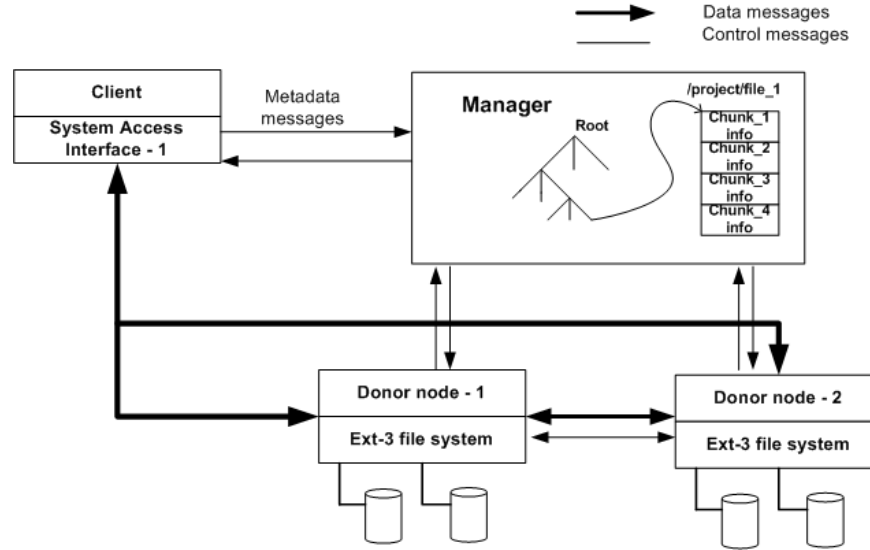


Figure 4: MosaStore storage system architecture. The figure highlights the three high-level components the system access interface (SAI) sitting at the client; the manager that stores all system metadata, and the donor nodes that store data chunks.

The metadata manager maintains the entire system metadata including but not limited to: donor nodes' status, file chunk distribution, access control information, and data object attributes. The metadata service is completely decoupled from the data service to provide high scalability. Each file is divided into fixed size chunks that are stored on the storage nodes. The mapping between a file and corresponding chunks are stored in the manager as a part of chunk distribution. Further, the Manager uses a round robin chunks placement policy hence when a new file is created on a stripe of n nodes the file's chunks are placed in a round robin fashion across these n nodes.

The donor nodes contribute storage space (memory or disk based) to the system. Donor nodes serve clients' chunk store/retrieve requests and also interact with the manager by publishing their status using a soft-state registration process.

The system access interface (SAI) is a collection of implemented FUSE call-backs that operate on a client node and provides a mechanism to access the storage space offered by the donor nodes. The SAI implementation supports a few important POSIX file system calls including sequential read and writes.

To MosaStore, we added garbage collection using epidemic protocol (make it efficiently use the scarce storage space – otherwise the application / storage will crash in supercomputers due to lack of memory), mechanisms to support POSIX file API (importantly random reads and writes, flush, delete – real workflow applications use these APIs), and modules to support chain and parallel replication (to reduce the impact of hot spot files), data placement policy and client side caching in order to build an intermediate storage system. Further we also made the intermediate storage system more configurable by exposing control knobs in configuration files.

Finally we successfully integrated the intermediate storage with workflow runtime and evaluated with synthetic and real workflow applications. The evaluation (§3.3) shows that the system successfully supports the workflow applications using all the above features in Blue Gene/P platform.

This intermediate storage is currently used by several researchers in University of Chicago and Argonne National Laboratory.

3.3 Intermediate Storage Evaluation

The intermediate storage (i.e. enhanced version of MosaStore in §3.2 [17]) was integrated with the workflow runtime Swift [6] and evaluated with both synthetic and real workflow

applications. The synthetic application is fully data intensive and used to obtain an estimate on the upper bound on the achievable gain. Workflow applications are realistic and more complex; have both computation and storage I/O and uses variety of POSIX file system API (file creation / deletion, read, write and etc).

The experiments were performed on a miniature version of Blue Gene/P experimental platform with 1024 nodes. This platform is exactly similar to the one described in section §2.1.2 and has two dedicated central storage instances, one with PVFS and other with GPFS. In all the experiments the MosaStore manager was deployed on one of the compute nodes and the rest of the nodes are used to run the storage nodes, the MosaStore SAI, and application. During these evaluations the interference from other users was negligible since the machine was not used by other users.

The ability to successfully run two complex workflow applications (Montage [2] and modFTDock [1]) and synthetic application with Swift [6] and the intermediate storage on Blue Gene/P with 128 node validates that the intermediate storage works functionally correct. And implies the intermediate storage system can be deployed during application launch time, supports minimal POSIX API required by the workflow applications (otherwise applications would have failed) and also has a successfully functioning garbage collection (otherwise applications would have crashed due to lack of memory).

We did a complete evaluation of the intermediate storage with synthetic benchmark (presented in §3.3.1) and modFTDock only (presented in §3.3.2). The evaluation presented in §5.2 validates that the intermediate storage function correctly with Montage and other synthetic benchmarks.

3.3.1 Synthetic Benchmark

The workload: We ran data-intensive application pipelines (proportional to number of compute nodes) in parallel (Figure 5) using Swift. Each of these pipeline stages-in a common input file from the shared GPFS central storage system and then goes through three processing stages that read input of 10 MB from the intermediate store and write the intermediate output of 20MB to the intermediate store, then the final output of 1MB is staged out back to the central storage system.

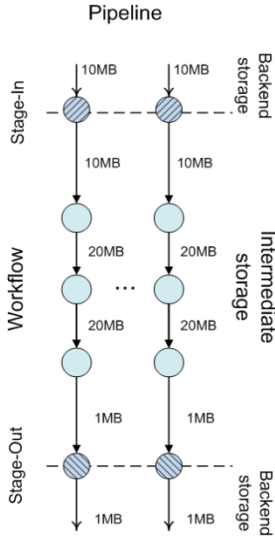


Figure 5: Pipeline

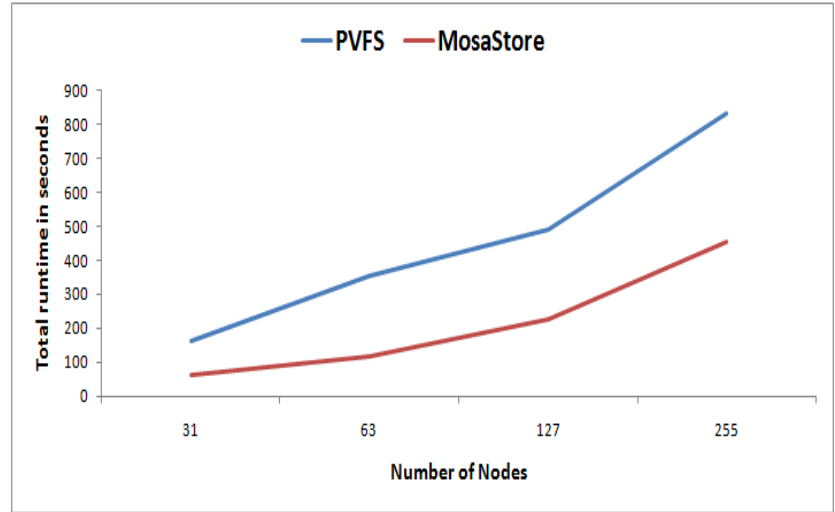


Figure 6: Pipeline runtime on BG/P

Evaluation Results: Figure 6 presents the performance of pipeline benchmark on Blue Gene/P. We compare the intermediate storage (labeled as MosaStore) with the central storage system PVFS [50]. *Our intermediate storage performs 100% better than central storage system.* The performance gain comes from both the ample network bandwidth available between compute nodes and the minimal POSIX file system interface implementation supported by the intermediate storage. Here it is important to notice that both systems have scalability issues with increasing number of nodes. Our investigation showed that in the case of intermediate storage

system scalability issues come from two components: first the Swift runtime has scalability issues and second the single manager on MosaStore does have its own limitations. We believe that distributing the MosaStore manager will provide better performance and scalability at large scale. *Further it is important to notice that the benchmark is fully data-intensive and for real applications the performance gain may be less than the results reported here.*

3.3.2 modFTDock

We ran modFTDock [1] at larger scale on BG/P to verify scalability and explore whether the performance gains are preserved when compared to much more powerful backend storage (GPFS) available on this platform. Figure 7 shows the modFTDock runtime on BG/P while varying the number of nodes allocated to the application. The workload size increases proportionally with the resource pool.

On the one side, we notice a consistent 20-40% performance gain of our intermediate storage over GPFS. On the other side, we would like to highlight that modFTDock is a compute bound applications and an intermediate storage system with minimal POSIX support will bring much large gains than this for data intensive applications.

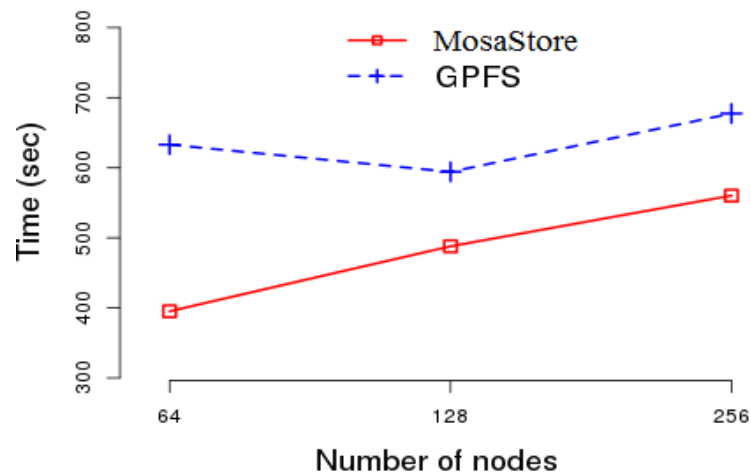


Figure 7: modFTDock runtime on BG/P

4. A Case for Workflow-Aware Storage Systems

This chapter explores one possible way to improve the performance of intermediate storage system for workflow applications. A workflow-aware storage system which permits specialized per-file optimizations and exposes data placement information can efficiently support the workflow applications as they generate regular data flow patterns.

The rest of the chapter builds a case for a workflow-aware storage system by identifying the opportunities. The next chapter quantifies the benefits of a workflow-aware storage system with an experimental study.

This chapter argues that there is still room left to improve the performance of workflow applications (§4.1) and provides evidence for data access patterns in workflows and recommends possible storage level optimizations for each pattern (§4.2). Finally the chapter highlights multiple ways to identify the data access patterns and provides evidences to show that these optimizations can be incorporated in a storage system without increasing its complexity (§4.3).

4.1 The Opportunities

Two observations reveal that specializing an intermediate storage system will bring promising gains for workflow-based applications: First, the workflows are composed of basic data access patterns. These patterns render existing storage systems unable to harness all optimization opportunities as this often requires enabling conflicting optimizations or even conflicting design decision at the storage system level. Second, when scheduling, most workflow runtime engines make suboptimal decisions as they lack detailed data location information that is generally hidden by the storage system.

An intermediate storage system that aggregates the resources of the computing nodes (e.g., disks, SSDs, and memory) provides two key advantages. First, it can be efficiently

configured to support the data access patterns generated by workflows through the file or directory level optimizations. Second, an intermediate storage system can expose the data placement information so that the workflow runtime engine can make data-aware scheduling decisions (Figure 8).

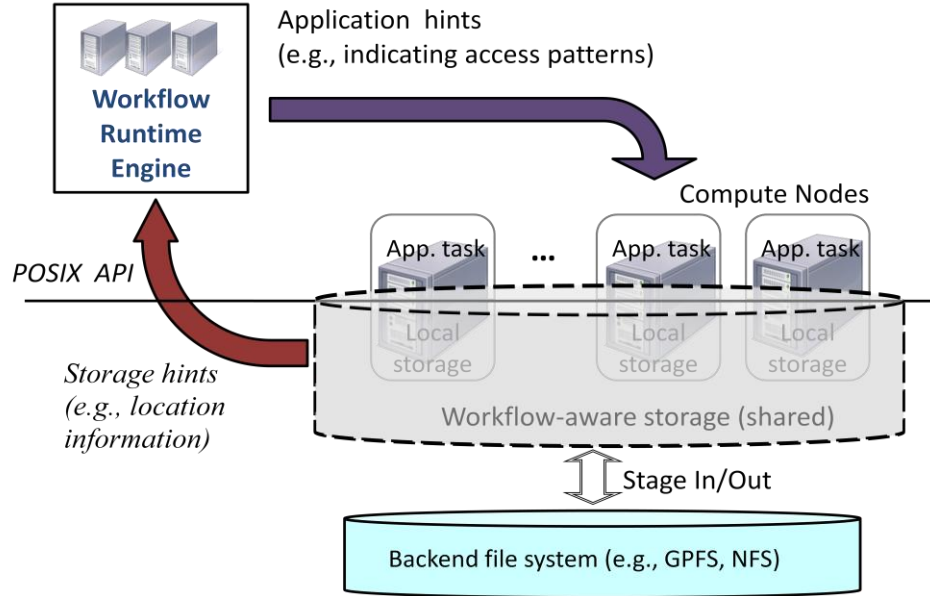


Figure 8: A case for workflow aware storage system

To support specific data access patterns, the storage system will use hints [12] that will drive the data layout (e.g., co-placement, replication levels, chunk placement) that indicate the expected access pattern. Section §4.3 argues 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. In addition the extended POSIX interface can be used by the storage to expose the data placement information to a workflow runtime engine.

Building a workflow aware storage system is indeed feasible and can bring significant performance gains. It is feasible, for two reasons: First, previous studies showed that workflows have a small set of common data access patterns, thus a small set of storage optimizations are

enough to serve these patterns. Second, our study shows that these optimizations can be incorporated in a high-performance storage system without significantly increasing its complexity (§4.2).

Finally it is important to state that a number of alternative approaches (§2.2) have been proposed to alleviate the storage bottleneck for workflow applications. They range from storage glide-ins (e.g., BADFS [36]) to building application-optimized storage systems (e.g. HDFS [33], BADFS [36]), to building a configurable storage system that is tuned at deployment time to better support a specific application [40], to offering specific data access optimizations (e.g., location-aware scheduling [51], caching, and data placement techniques [52]). Taken in isolation, these efforts do not fully address the problem the workflow applications face presently 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. The goal is to integrate lessons of the above past work in the context of workflow application and recommends a set of techniques to improve the performance of workflow applications.

4.2 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), Katz et al. [20] (12 applications), Shibata et al. [53] (5 applications), Bharathi, et al. [11] (5 applications) and Ustunet et al. [54]. This section starts from the workflow data access patterns identified by the above studies. *Further this section extends the already available data access patterns with a set of identified new data access patterns scatter, gather and distribute. Even though scatter, gather and distribute patterns already exist in several other areas such as*

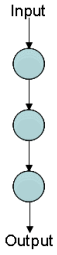
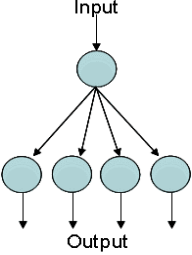
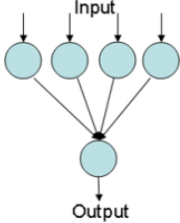
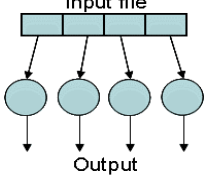
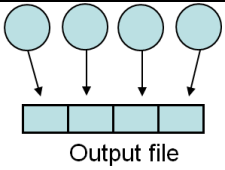
MPI, it is important to notice that these patterns are new in the context of many-task workflow applications. Further it proposes the file-level / data-object level optimizations that a storage system needs to support to improve the performance of each pattern (summarized in Table 1). The rest of the section briefly presents the common data access patterns in the workflows and the opportunities for storage optimizations at the data-object level.

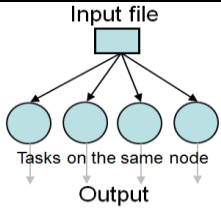
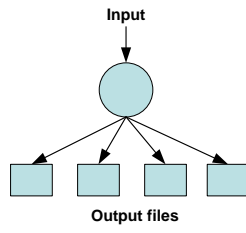
- **Pipeline:** A set of compute tasks are chained in a sequence such that the output of one task is the input of the next task in the chain. An optimized storage system can store the intermediate files on the same machine as the one that executes the task (if space is available) to increase access locality and to efficiently use 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 processes a set of 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:** A single file is read by multiple compute nodes where each node accesses a disjoint ‘region’ in the file. An optimized object storage system can configure the chunk size to be smaller than the region size (such that no two regions share a chunk) and

optimize its operation by placing all the chunks that belong to a particular region on the same node, placing different file regions on different nodes, and by exposing chunk-level data placement to the application.

- **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 chunk size to be smaller than that the region size (such that no two regions share a chunk) and to provide metadata support for multiple concurrent updates to a single file metadata, enabling parallel writes to all file regions. The data placement is optimized based on the next step in the workflow (e.g. single node for pipeline, or on multiple nodes for scatter).
- **Reuse:** A single file is read multiple times by one or multiple tasks scheduled on the same compute node. An optimized storage system can replicate the file onto a storage node on the same machine as the one that executes the tasks (if such storage node exists and space is available) or cache a copy of the file at the same node.
- **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 distribute pattern. Such storage will spread the files in a balanced way across storage nodes and collocate the chunks belongs to a file on a single node or on minimal set of nodes. Also based on the application it may replicate the files.

Table 1: Popular data access patterns generated by workflows. Circles represent computations. An outgoing arrow indicates 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. We use a notation similar to that used by Wozniak et al. [9].

Pattern	Pattern Details	Optimizations
Pipeline		<ul style="list-style-type: none"> • Node-local data placement (if possible). • Caching. • Data-informed workflow scheduling.
Broadcast		<ul style="list-style-type: none"> • Optimized replication taking into account the data size, the fan-out, and the topology of the interconnection.
Reduce		<ul style="list-style-type: none"> • Reduce-aware data placement: co-placement of all output files on a single node. • Data-informed workflow scheduling.
Scatter		<ul style="list-style-type: none"> • Application-informed chunk size for the file. • Application-aware chunk placement. • Data-informed workflow scheduling.
Gather		<ul style="list-style-type: none"> • Application-informed chunk size for the file. • Application-informed chunk placement.

Pattern	Pattern Details	Optimizations
Reuse		<ul style="list-style-type: none"> • Application-informed replication. • Application-informed caching.
Distribute		<ul style="list-style-type: none"> • Application informed file and chunk placement. • Application informed replication.

4.3 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: the workflow runtime engine can provide this information 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 I/O operations. This section briefly describes 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 be provided to the underlying storage system to optimize its operations based on these hints. Santos-Neto et al. [12]

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-tune the optimizations depending on the platform or the application (e.g., how many replicas is the optimal, what is the optimal cache size). Past work [55] uses a monitoring module to collect information on the access patterns and predicts the future ones. The storage system can automatically 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 in our group [13].

Application profiling: Another approach to fine-tune 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.

5. Workflow-aware storage system: An Opportunity Study

The previous chapter provided arguments that building a workflow aware storage system is indeed feasible. This chapter evaluates the potential gains a workflow-aware storage system can bring using micro-benchmarks, application-level synthetic benchmarks and scientific workflow applications.

The 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, the ability to expose data location through POSIX's extended file attributes was added to MosaStore storage system. This enables MosaStore to be integrated with a workflow runtime engine that supports data-aware scheduling. The chapter explains the hacks (customizations) introduced in MosaStore for each pattern (§5.1) followed by detail evaluation (§5.2).

5.1 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 helps 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 the enhanced version of MosaStore (i.e. the intermediate storage system in §3.2) was customized according to the pattern. This section briefly presents these hardcoded customizations (described in more detail in §5.2.2 in the context of the evaluation experiments). The 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. All optimizations described below harness the fact that MosaStore exposes data placement through POSIX-extended file attributes and 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.** The MosaStore data placement module was changed to prioritize storing output files produced by a pipeline stage at the node where the task corresponding to that stage runs. If data does not fit on the local node, then the file's chunks are shipped remotely through the normal MosaStore mechanisms.
- **Optimized data placement for the reduce pattern.** The MosaStore was changed 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 chunks 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. The default MosaStore lazy replication mechanism was changed to eager parallel replication: replicas are created eagerly while each chunk is written to storage.
- **Optimized data chunk placement for the scatter and gather patterns.** Unlike other patterns described above that require optimizations at the file level, scatter and gather require chunk-level optimizations, as a single file's chunks are accessed by a number of

compute tasks in parallel. Consequently, we set the MosaStore chunk size to match the application per-node access region size, and constrain the MosaStore data placement such that we can determine where each chunk of a file is placed. Further, we optimize the scheduling decision to run the compute task on the node that has the specific file chunk accessed by that task.

In addition to the per-pattern customizations described below, one general optimization was applied in all the experiments: local file access was prioritized over the remote access to take advantage of access locality. The storage client was changed to prioritize reading chunks directly from the local storage node instead of reading from remote nodes (if a chunk is available).

5.2 Evaluation

The evaluation of workflow-aware storage system is done in three ways. First to quantitatively evaluate its impact in realistic settings we designed a micro benchmark to evaluate the cost of accessing local vs. remote storage node to serve application's data requests (§5.2.1). Second, the application-level synthetic benchmarks (§5.2.2) are designed to mimic the data access pattern of the scientific workflows. Since the real scientific workflows are complex and often have multiple I/O patterns with several stages (§4.2) using the synthetic benchmarks will be a good method to quantify the gains for each pattern. Finally, we use Montage [2] and modFTDock [1] – real applications to evaluate the workflow-aware storage system.

The 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 0.5x to 3x 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 its well provisioned hardware.)

5.2.1 Micro Benchmark: The Impact of Locality

Experiment setup: We deployed the MosaStore storage system with the manager, one storage node and one SAI in two different setups: First, to evaluate the cost of accessing a local file, we deploy the storage node and the SAI on the same machine. Second, to evaluate the cost of accessing remote files, we deployed the storage node and SAI on two different machines. In both setups the manager was deployed on a separate machine to keep the metadata cost constant across the experiments. 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.

Customizations: The default MosaStore system uses regular sockets [56] to communicate between the storage nodes and the SAI. The regular socket uses the standard network stack; hence, it adds an additional overhead when the SAI and the storage node are collocated on the same physical node. We changed the MosaStore to use domain sockets [56] and partially eliminate this overhead in this situation. The reason is that domain sockets use shared memory to communicate instead of the network stack, while, at the same time, support the standard socket APIs.

The workload: The micro benchmark sequentially writes 30 files of 1 GB via a single SAI and then sequentially reads these files. We chose large files and write/read the files back to back to reduce the effect of caching especially when data reside on disk. The benchmark reports the write/read throughput.

Evaluation results: We evaluated the achievable performance gain due to locality while having the data chunks stored on either *RAMdisk* or *spinning-disk*. We present, in Figure 9, the I/O throughput for the following configurations: the local storage node when using the domain socket

(labeled ‘Domain’ in the figure), the local storage node with regular socket (‘*Regular*’), and the remote storage node with regular socket (‘*Remote*’). Additionally, for comparison, we present the results of running the same benchmark when using the native file systems (*ext3* on *spinning-disk* and *tmp-fs* on *RAMdisk* - labeled as ‘*Local*’ in the figure) which represent ideal baselines, and eliminate all MosaStore overheads.

Figure 9 presents the I/O throughput when the storage node is backed by *spinning-disk* (left plot) and *RAMdisk* (right plot). For each plot there are two sets of columns presenting the write and, respectively, the read throughput. We make the following observations. When data chunks are stored on spinning-disk, locality does not have a pronounced impact on the read throughput; the reason is that in this case the disk itself is the bottleneck (Figure 9). Locality, however, provides significant performance gains for writes, even when data chunks are stored on disk. This is because the writes often hits the file system cache hence the network becomes the bottleneck. When the storage node is backed by a *RAMdisk*, the network become bottleneck in both cases and both local read and writes are much faster than remote read and remote write.

Further, in most cases, accessing local data through domain sockets offers a performance advantage. Compared to accessing local data through the regular TCP sockets, domain sockets offer 27% - 47% (on *RAMdisk*) and 6%-10% (on *spinning-disk*) higher throughput in the four configurations we study. As expected, accessing data stored on a remote node leads to a throughput 52% to 84% lower (except in the read from spinning disk case mentioned above). The performance penalty is magnified when the storage nodes are backed by *RAMdisk* instead of spinning-disks.

Finally, this experiment allows us to have a first estimate of the overheads added by MosaStore when compared to a local storage system. While these overheads appear significant, we note that the comparison is not entirely fair: we compare a distributed file-system (deployed

such that some components, the manager in our case, are indeed remote) with a local file-system. As expected when the storage node is backed up by the much faster *RAMdisks* the throughput loss is much more pronounced than when the storage node is backed up by spinning disk (up to 5.2x throughput loss for *RAMdisk* vs. up to 1.06x throughput loss for *spinning-disk*).

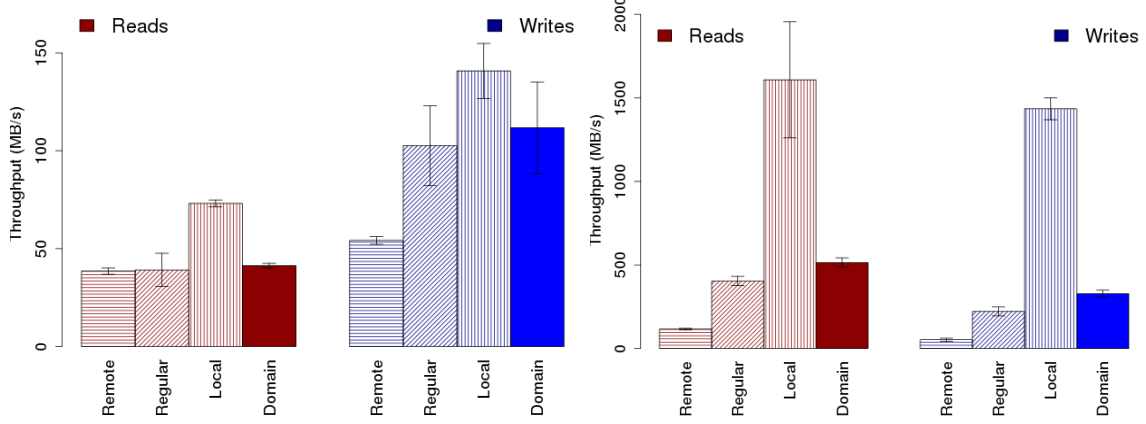


Figure 9: I/O throughput when the storage node is backed by *spinning-disk* (left plot) and *RAMdisk* (right plot). For each plot there are two sets of columns presenting the write and, respectively, the read performance. Note that the axes use different scales in the two plots. Figures represent average throughput, and standard deviation in error bars, over 30 reads/writes.

5.2.2 Synthetic Benchmarks

We evaluate our approach using a set of application-level synthetic benchmarks. We designed the benchmarks to mimic the data access pattern of the scientific workflows. These benchmarks evaluate the impact of pattern specific storage optimizations. We evaluate the synthetic benchmarks on storage nodes supported by either *spinning-disk* or *RAMdisks*.

Experiment setup: Current workflow processing often works as follows: workflow applications 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, persisting them again on the back-end storage system. The intermediate shared storage is faster than back-end storage and provides a high performance scratch space to the application.

Our experiment setup is similar to this scenario. Throughout the evaluation, we compare the performance of the following intermediate shared 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 back-end 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 Lustre. Further, although NFS is not typically used in large scale platforms, at our scale with the setup of 20 machines, it fairly approximates a well-provisioned shared back-end storage system.

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. 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.

The sets of synthetic application benchmarks fit the standard workflow application model (stage-in, workflow execution and stage-out) and are composed of read/write operations that mimic the file access patterns described earlier. The benchmarks are purely I/O bound and provide an upper bound on the achievable performance for each pattern. For this opportunity study, we looked at several real world workflow applications [10, 16, 17, 18, 19] and selected three workload types with different file sizes. Figure 10 summarizes these application benchmarks.

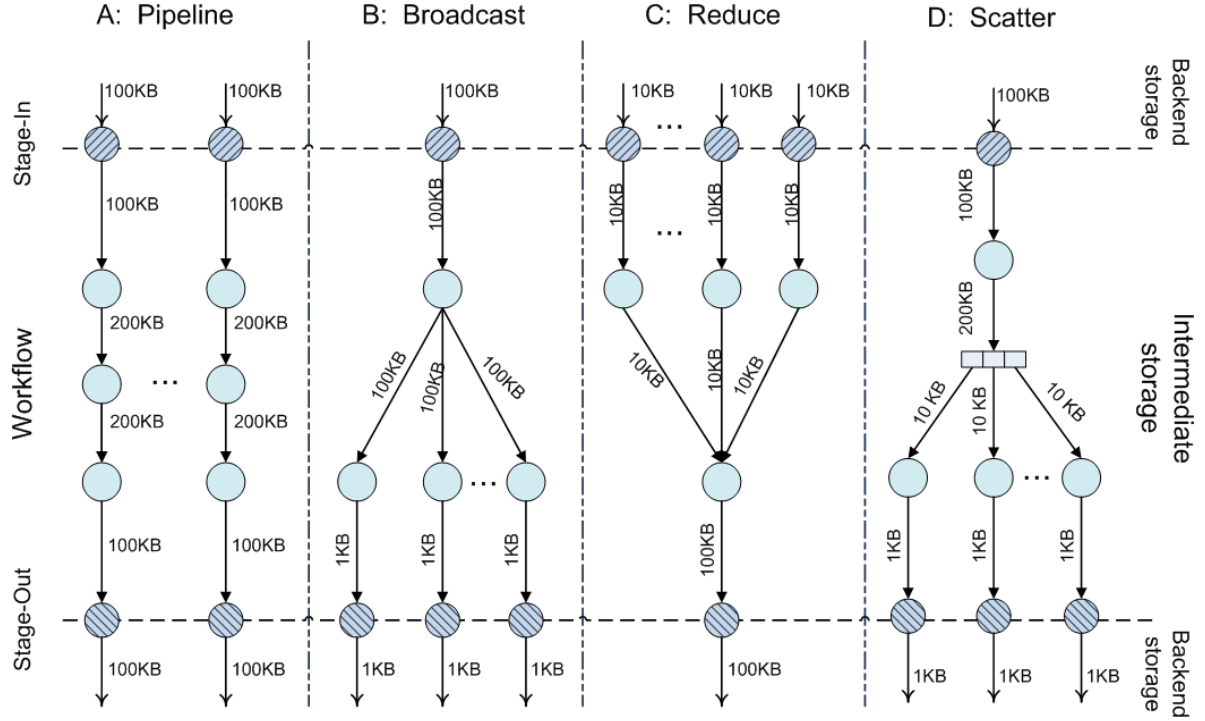


Figure 10: Summary of synthetic benchmarks for pipeline, broadcast, reduce, and scatter patterns. Nodes represent workflow stages (or stage-in/out operations) and arrows represent data transfers through files. Labels on the arrows represent file sizes for the ‘small’ workload. The other workload sizes are presented in Table 2.

Table 2: File sizes for different workflow patterns.

Data access patterns	Workloads (file size for input, intermediate & output)		
	Small	Medium	Large
Pipeline	100KB, 200KB, 10KB	100 MB, 200 MB, 1MB	1GB, 2GB, 10MB
Broadcast	100KB, 100KB, 1KB	100 MB, 100MB, 1MB	1 GB, 1GB, 10 MB
Reduce	10KB, 10KB, 200KB	10MB, 10MB, 200 MB	100MB, 100MB, 2 GB
Scatter	100KB, 190KB, 1KB	100 MB, 190MB, 1MB	1 GB, 1900MB, 10 MB

The rest of this section presents, for each synthetic benchmark: pipeline, broadcast, reduce, and scatter, the detailed experiments we executed, the MosaStore customizations that support them, and the performance evaluation results.

5.2.2.1 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 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 10 – A): We run in parallel 19 application pipelines similar to the ones described in the Figure 10. 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.

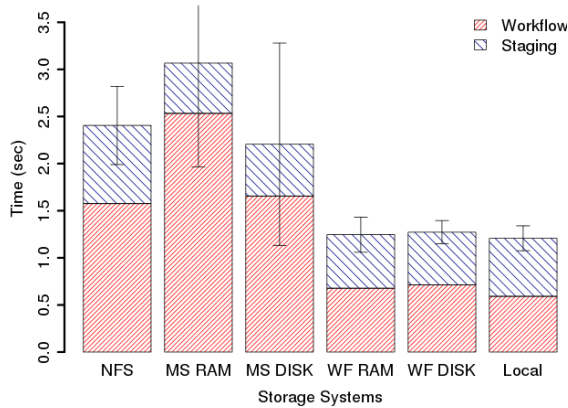


Figure 11. Pipeline pattern – small files. Average execution time (in seconds) for small file sizes. Error bars represent standard deviation for all stages of the workflow (the entire experiment time).

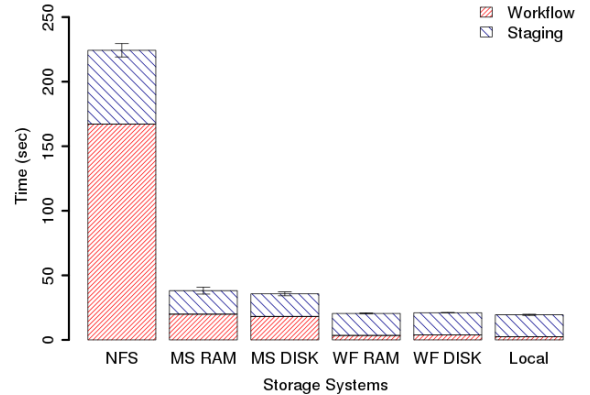


Figure 12. Pipeline pattern – medium files. Average execution time (in seconds) for medium-size file. Error bars represent standard deviations for the entire experiment.

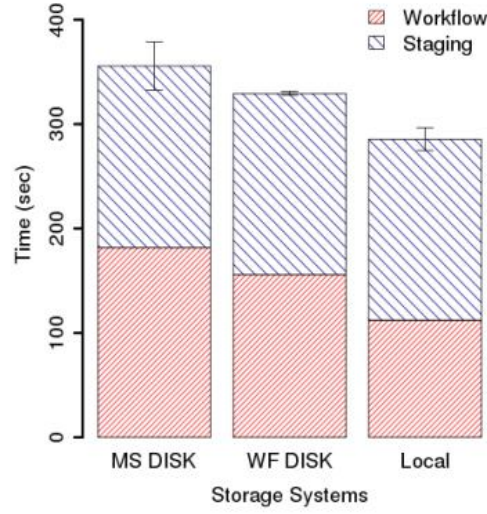


Figure 13. Pipeline pattern large files. Average execution time (in seconds) for large file sizes.

Evaluation Results: Figure 11, Figure 12 and Figure 13 present the performance of our systems for small, medium, and large workloads. The figures present distinctly (as stacked bars) the performance for data staging (stage-in time plus stage-out time) and the performance for the pipeline stages that touch the intermediate storage system. We experiment with four possible intermediate storage configurations: (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 ‘MS RAM’ or ‘MS DISK’ depending on whether the storage nodes are backed by *RAMdisk* or *spinning-disk*); and (4) a MosaStore with modifications to become workflow aware (labeled ‘WFRAM’ or ‘WFDISK’).

We note that we could not execute the ‘large’ workload for three configurations: The NFS crashes (or takes unreasonably long time) under this workload and there isn’t enough space to execute this workload with RAM based storage nodes. 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, the workflow aware storage is 10x faster than NFS, and almost 2x faster than vanilla 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. Further with large files, we could not run the RAM disk experiments due to the memory limitation in our cluster and most part of the time is spent in staging phases.

The local configuration presents the optimal data placement decision for the pipeline pattern, serving as a baseline. In both experiments the workflow aware storage (‘WFRAM’ and ‘WFDISK’) lags behind the local storage due to added overhead of metadata operations and additional context switches and memory copies introduced by fuse user-level file system.

5.2.2.2 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 10 – B*): An input file is staged-in to the intermediate storage from the back-end storage (i.e., the NFS). Then the first stage of the benchmark reads the input file and produces a broadcast-file on the intermediate storage. In the second stage, the broadcast-file is read by 19 processes running in parallel on 19 different machines. Each of these processes writes its output independently on the intermediate storage. As a last stage, the output files are staged-out to the back-end storage in parallel.

Evaluation Results: Figure 14, Figure 15 and Figure 16 present the performance for this benchmark for ‘medium’ and ‘large’ workloads, while varying the number of replicas created. WF performs better than MStore (i.e., no replication), reaching the best performance for 8 replicas for medium files and 4 replicas for large files. This result matches the expectation of the potential benefits of WASS approach.

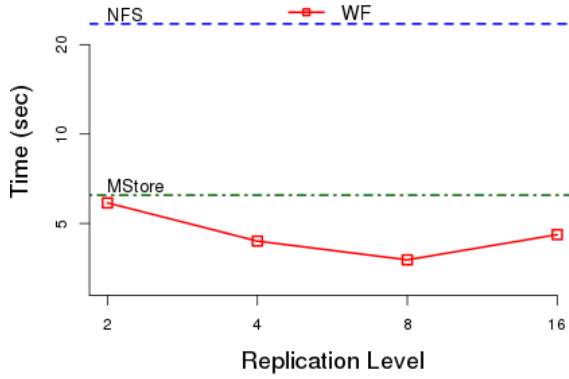


Figure 14: Average execution time for broadcast synthetic benchmark with medium workload. All storage systems are deployed on spinning disks.

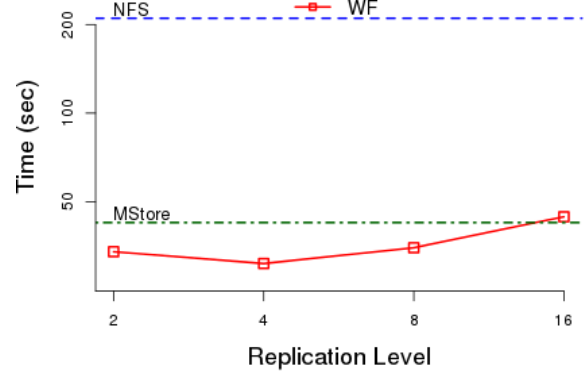


Figure 15: Average execution time for broadcast synthetic benchmark with large workload. All storage systems are deployed on spinning disks.

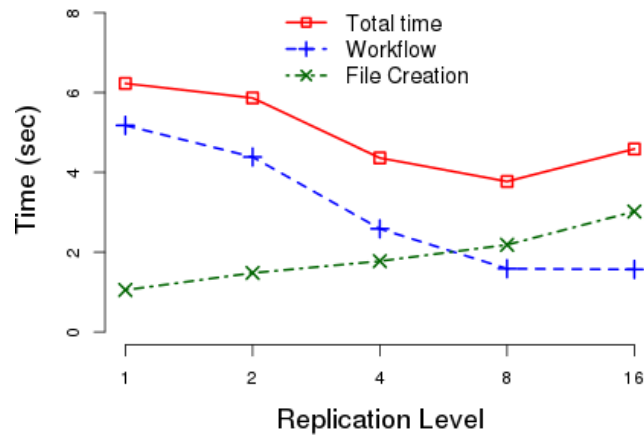


Figure 16. Breakdown of broadcast benchmark for the ‘medium’ workload.

For more replicas than this optimal number, the overhead of replication is higher than the gains of adding more data access points. A similar pattern can be observed for small files; in this case, replication does not pay off at all. To better understand the trade-off between adding more access points and creating extra replicas, Figure 16 shows the breakdown of the benchmark phases. As the number of replicas increases, the time to process the data (the ‘workflow’ line) decreases and the time to create the replicas increases.

5.2.2.3 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 10 – C): During the stage-in phase 19 input files are staged-into the intermediate storage from the back-end storage. In the first stage of the benchmark 19 executables, running in parallel on 19 different machines, each reads an input files and produce an intermediate file. In the next stage a single executable reads the intermediate files and produces the reduce-file (final output). The reduce-file is staged-out to the back-end store (the NFS).

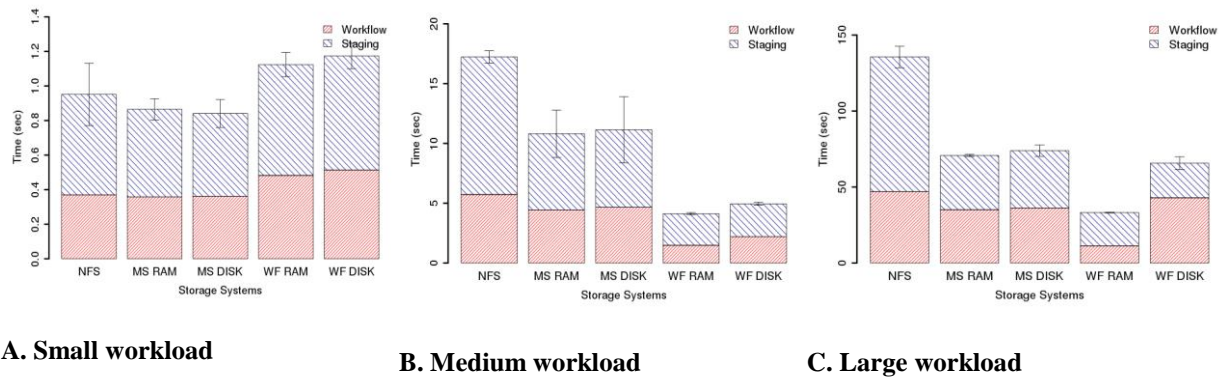


Figure 17: Reduce pattern. Average benchmark execution time (in seconds).

Evaluation Results: Figure 17: shows the benchmark runtime for all three workloads and the five different configurations of the intermediate storage system (intermediate storage on NFS, MosaStore and the workflow aware system and, for the last two options, with storage nodes using *RAMdisk* and *spinning disk*)

With *spinning-disk* configuration, for medium and large files, workflow-aware is between 3.9x (with large files) to 3.4x (with medium files) faster than NFS and 1.2x (for large files) to 2.25x (for medium files) faster than MosaStore default configuration. NFS performs relatively similarly to the other options for small files: this happens because the advantages offered by the faster intermediate system are cancelled by its additional overheads that start to dominate for small files.

For the large workload, workflow time on WF DISK is longer compared to MS DISK. This happens because during the reduction phase the data is on the spinning-disk and the disk throughput becomes a bottleneck given the concurrency of several clients in parallel to write the data. However, WF RAM has significantly shorter workflow time since the entire data is on *RAMdisk* without the throughput bottleneck of *spinning-disks*. With *RAMDisk* configuration, workflow aware storage system achieves the highest performance with medium and large files workload, up to 2.6x times faster than MS_RAM and up to 1.9x faster than WF_DISK. This is mainly due to a significant reduction in the workflow execution time. This reduction in workflow execution time is due to the optimized data placement in workflow aware storage that increases the data locality.

5.2.2.4 Scatter Pattern Evaluation

Customization: To efficiently support the scatter pattern, we applied two modifications to MosaStore: first, we enable configuring the storage system’s chunk size in order to match the application-level scatter ‘region’ size (i.e., the region of the file that will be read by a single application), and second, we modify the MosaStore data placement to colocate all the chunks belonging to a particular file region on the same node.

The workload (*Figure 10 – D*): Initially an input file is staged-in to the intermediate storage from the back-end storage (i.e., the NFS). The first stage of the workflow reads the input file and produces a scatter-file on intermediate storage. In the second stage, 19 processes running in parallel on different machines. Each process reads a disjoint region of the scatter-file and produces an output file. Finally, at the stage-out phase, the 19 output files are copied to the back-end storage.

Evaluation Results: In experiments with MosaStore and workflow-aware storage, the scatter benchmark spends equal amount of time in staging in the input file (12 seconds on average for the large workload) and creating the scatter file (27.5 seconds on average for the medium workload). The stage in time and file creation time are significant, amounting to 70-90% of the benchmark time. Staging time and scatter file creation time on NFS was significantly slower than the other systems. For clarity of the presentation we present only the runtime of the scatter stage (stage-2) in Figure 18 and Figure 19. Further, for small workload, the evaluation results were inconclusive due to high variance; hence we do not present them here. With *spinning-disk* configuration, for medium and large files, workflow-aware storage is around 8.1x faster than NFS and 1.5x faster than MosaStore default configuration. With *RAMDisk* configuration, workflow aware storage system achieves the highest performance with medium and large files workload, up to 10.4x times faster than NFS and 2x faster than MosaStore default configuration.

This is mainly due to the application customized data placement in the workflow aware storage system that significantly increases data access locality.

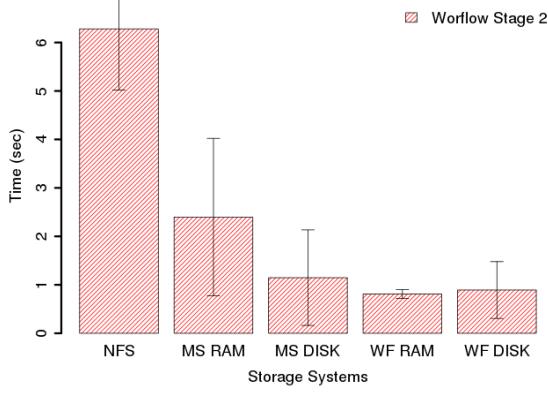


Figure 18. Scatter pattern medium files. Average execution time (in seconds) and standard deviation for the scatter stage of the benchmark (medium file sizes)

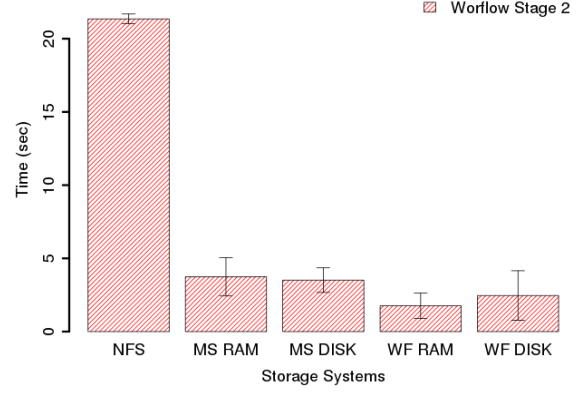


Figure 19: Scatter pattern large files. Average execution time (in seconds) and standard deviation for the scatter stage of the benchmark (large file sizes)

5.2.3 Workflow Applications

The previous section demonstrated that the benefits of workflow-aware storage system approach: optimizing for data access patterns with synthetic benchmark provide significant performance over an un-optimized storage. This section evaluates the promising gains of workflow-aware storage using a significantly more complex real workflow applications Montage (§5.2.3.1) and modFTDock (§5.2.3.2).

To run these experiments we used a set of 11 machines from the same cluster that we used in synthetic benchmarks. We deployed MosaStore/Workflow-aware storage manager on a dedicated node and used other 10 nodes to run the rest of the system (storage nodes and MosaStore SAI were co-deployed in all the 10 nodes). *We use a set of cross-layer mechanisms developed by Al-kiswany et al. [19] to optimize reduce patterns in these experiments.*

5.2.3.1 Montage

The Montage [2] workflow is chosen for two reasons. First it is a complex and data intensive workflow and second it is a popular application used by many others to evaluate the many-task platforms [27], [45]. The Montage workflow is composed of 10 different processing stages with varying characteristics (Table 3). The workflow uses the reduce pattern in 2 stages and the pipeline patterns in 4 stages (labeled in Figure 20). Nodes represent workflow stages and arrows represent data transfers through files.

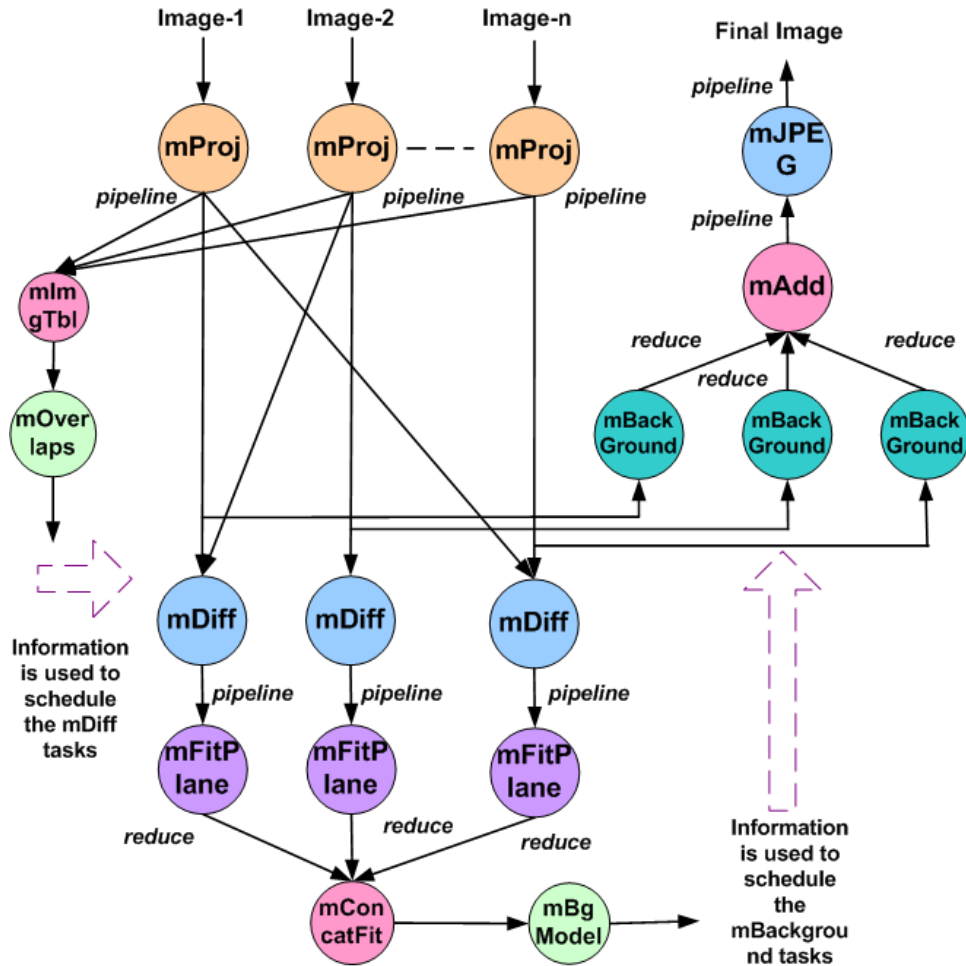


Figure 20: Montage workflow. The tags we use to indicate data usage patterns are presented in the figure. The characteristics of each stage are described in Table 3. Labels on arrow represent the data access patterns.

The I/O communication intensity between workflow stages is highly variable (presented in Table 3 for the workload we use). The workflow uses pyFlow framework as the workflow-runtime. Overall the workflow generates over 650 files with sizes form KB to over 100MB and about 2GB of data are read or written from storage.

Table 3: The characteristics of each stage for the Montage workflow

Stage	Data	#files	File size	Optimization
stageIn	109 MB	57	1.7 MB - 2.1 MB	
mProject	438 MB	113	3.3 MB - 4.2 MB	Yes
mImgTbl	17 KB	1		
mOverlaps	17 KB	1		
mDiff	148 MB	285	100 KB - 3 MB	Yes
mFitPlane	576 KB	142	4.0 KB	Yes
mConcatFit	16 KB	1		
mBgModel	2 KB	1		
mBackground	438 MB	113	3.3 MB - 4.2 MB	Yes
mAdd	330 MB	2	165MB	Yes
mJPEG	4.7 MB	1	4.7 MB	Yes
stageOut	170 MB	2	170 MB	Yes

Evaluation Results: Figure 21 shows the total execution time of the Montage workflow in five configurations: over NFS, and with MosaStore (labeled as MS-DISK / MS-RAM) and workflow-aware storage (WF-DISK / WF-RAM) deployed over the spinning disks or RAM-disks of local nodes. The workflow-aware storage system achieves the highest performance when deployed on disk or RAM-disk. When deployed on disk the workflow-aware storage achieves 20% performance gain compared to NFS. Further the workflow-aware storage achieves up to 10% performance gain compared to MosaStore when deployed on disk or RAM-disk. When deployed on disk the workflow-aware storage achieves 20% performance gain compared to NFS. Further

the workflow-aware storage achieves up to 10% performance gain compared to MosaStore when deployed on disk or RAM-disk.

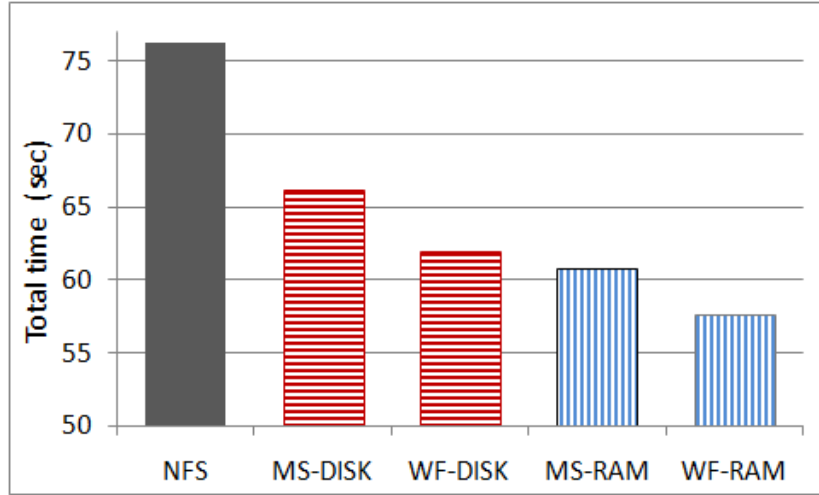


Figure 21: Montage workflow total execution time. Note that, to better highlight the differences, y-axis does not start at zero.

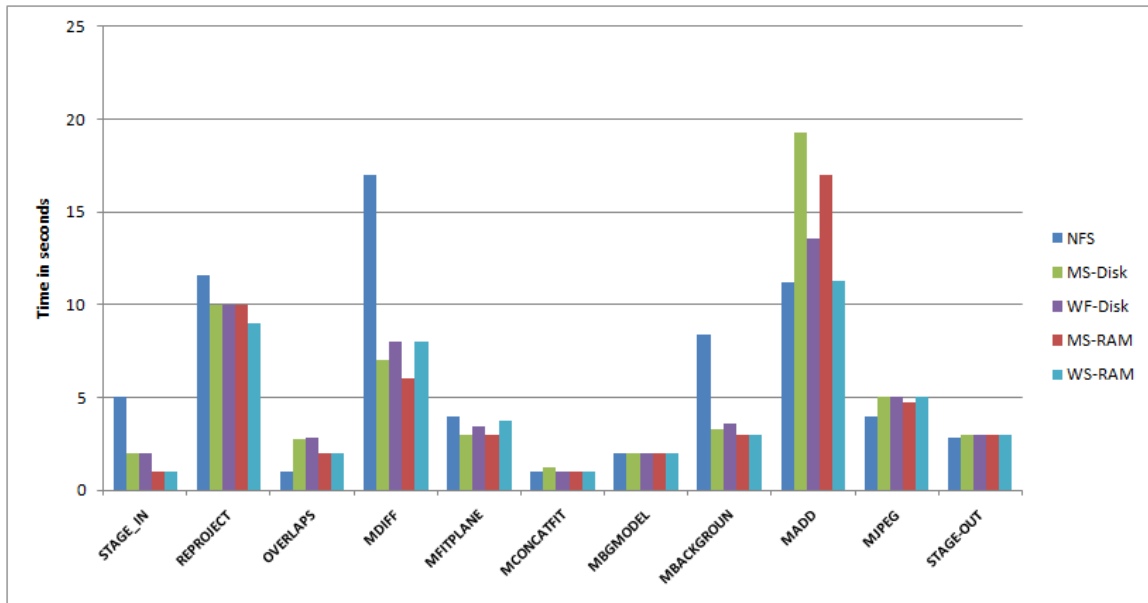


Figure 22: Montage workflow per-stage execution time

To quantify the amount of gain achieved by each optimization we analyzed the per-stage execution time (Figure 22). From Figure 22 we can clearly observe that the performance gain comes from the optimizations (optimizations for reduce pattern) used in *mAdd* and *reproject* stages. In all other stages workflow-aware storage system performs almost equal to MosaStore. Further investigations reveal this observation. The workflow-aware storage system may not bring performance gain for small files. This is because the cost of enabling the optimization and getting a file location from the workflow-aware storage may outweigh the achievable performance.

5.2.3.2 modFTDock

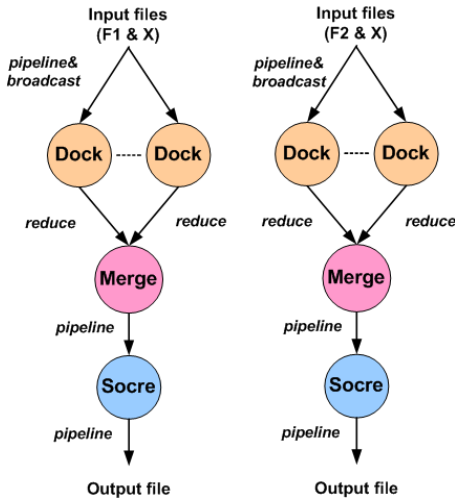


Figure 23: modFTDock workflow. Labels on arrows represent the data access patterns.

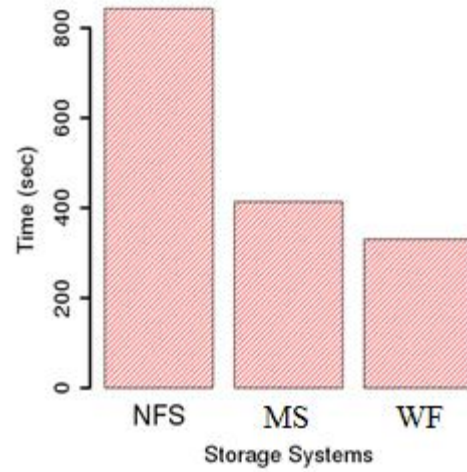


Figure 24: modFTDock workflow total execution time.

For modFTDock we use Swift to drive the workflow: Swift schedules each application stage, and tags the files according to the workflow pattern. As modFTDock combines the broadcast, reduce and pipeline pattern. The database is replicated (broadcast pattern) and the output of every dock stages is collocated on a single storage node that will execute the merge stage (reduce). The

merge output is placed on local storage node in order to execute the score stage on the same machine (pipeline pattern).

modFTDock experiments were run on cluster with 10 dock processes process the input files (100-200KB) and a database (100-200KB). The storage nodes are mounted on RAM-disks. Figure 24 presents the total execution time for the entire workflow including stage-in and stage-out times for MosaStore and workflow-aware storage. The workflow-aware storage optimizations enable a faster execution: modFTDock with Swift is 20% faster when running on workflow-aware storage than on MosaStore, and more than 2x faster than when running on NFS.

6. Conclusion

In this thesis we have demonstrated possible ways to improve the performance of workflow applications on large scale machines. In particular we have proposed a two-step solution to alleviate the performance of workflow applications and qualitatively evaluated the gains with both synthetic and real workflow applications.

First, we proposed and designed an intermediate storage system to increase the performance of workflow applications on large-scale machines. An intermediate storage system aggregates the storage resources on compute nodes and supports a minimal set of POSIX API will provide high performance and scalable storage space compared to a regular shared storage system. Second we explore the viability of a workflow-aware storage system: the opportunity of tuning the intermediate storage system depending on the data access pattern of a workflow application. To this end, we discuss the feasibility of building a workflow-aware storage system that can provide per-file optimizations, and experimentally evaluate the attainable performance gains with a workflow-aware storage system.

The evaluation with synthetic and real workflow applications highlights the significant performance gains achievable by both an intermediate storage system and a workflow storage system. An intermediate storage can bring up to perform 2x performance gain compare to a central storage system. A workflow-aware storage system (i.e. an intermediate storage optimized for application's data access pattern) can bring up to 3x performance gain for some data access patterns compared to a regular intermediate storage system.

Our findings highlight that a workflow-aware intermediate storage system is a promising direction to provide a high-performance scalable storage system for workflow applications.

Our research has two implicit assumptions. First, the workflow applications are data-intensive. Even though most of the many-task workflow applications are data-intensive, we

would like to point out that an intermediate storage system or a workflow-aware storage system may not provide noticeable performance gain for compute intensive applications.

Second, our workflow-aware system approach harness the fact that storage nodes and storage clients are always co-deployed on a cluster. Although this assumption may not be true in some systems, we would like to highlight that this is the case for large scale computer systems such as Blue Gene/P and Jaguar [15].

6.1 Future Work

Two research avenues are resulted from this research: First, one can build a workflow-aware intermediate storage system to demonstrate the potential gains highlighted by this study and second, exploring the feasibility of determining the workflow applications' data 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 [19], and by building workload monitoring and access prediction components and using predictions for storage system auto-tuning [14].

Here we also would like to highlight a few implications of our future research. First, a workflow-aware storage system approach may add some level of complexity to a storage system. However our experience during the opportunity study highlights that the complexity added by workflow-awareness is manageable and actually pays-off. Second, in realistic settings a workflow-aware storage system may not bring noticeable performance gain for some applications. A workflow-aware storage system brings two additional overheads: the cost of enabling storage-level optimizations and the cost of accessing the location of files. In some cases, these overheads may outweigh the performance gain that can be brought by storage

optimizations. However one can optimize these overheads and improve the performance of a workflow-aware storage system.

References

- [1] “modFTDock,” vol. 2012. .
- [2] A. C. Laity, N. Anagnostou, G. B. Berriman, J. C. Good, J. C. Jacob, D. S. Katz, and T. Prince, “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*, vol. 106, no. 3, 2009.
- [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,” *IEEE Workshop on Many-Task Computing on Grids and Supercomputers* . 2008.
- [6] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, “Swift: A language for distributed parallel scripting,” *Parallel Computing*, 2011.
- [7] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, “Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems,” *Journal of Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.
- [8] “Makeflow,” vol. 2012. .
- [9] J. Wozniak and M. Wilde, “Case studies in storage access by loosely coupled petascale applications,” *Workshop on Petascale Data Storage*, p. 16, 2009.
- [10] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, and M. Wilde, “Design and Evaluation of a Collective IO Model for Loosely Coupled Petascale Programming,” *Science*, 2008.

- [11] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of Scientific Workflows," *Workshop on Workflows in Support of Large-Scale Science*. 2008.
- [12] E. Santos-Neto, S. Al-Kiswany, N. Andrade, S. Gopalakrishnan, and M. Ripeanu, "Enabling Cross-Layer Optimizations in Storage Systems with Custom Metadata," in *ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC) - Hot Topics Track*, 2008.
- [13] L. B. Costa and M. Ripeanu, "Towards Automating the Configuration of a Distributed Storage System," 2010.
- [14] S. A.-K. and M. R. Lauro Beltrão Costa, Abmar Barros, Emalayan Vairavanathan, "Predicting Intermediate Storage Performance for Workflow Applications," *Submitted to CCGrid*, 2013.
- [15] I. B. M. B. G. team, "Overview of the IBM Blue Gene/P Project," *IBM Journal of Research and Development*, vol. 52 , 2008.
- [16] E. Vairavanathan, S. Al-Kiswany, L. Costa, Z. Zhang, D. Katz, M. Wilde, and M. Ripeanu, "A Workflow-Aware Storage System: An Opportunity Study," in *International Symposium on Clusters, Cloud, and Grid Computing (CCGrid)*, 2012.
- [17] H. Y. and M. R. Samer Al-Kiswany, Emalayan Vairavanathan, Lauro Beltrão Costa, "MosaStore functional and design specification," 2012.
- [18] and M. R. Emalayan Vairavanathan, Samer Al-Kiswany, Abmar Barros, Lauro Beltrão Costa, Hao Yang, Gilles Fedak, Zhao Zhang, Daniel S. Katz, Michael Wilde, "A case for Workflow-Aware Storage: An Opportunity Study using MosaStore," *Submitted to FGCS Journal*.
- [19] H. Y. and M. R. Samer Al-Kiswany, Emalayan Vairavanathan, Lauro Beltrão Costa, "The Case for Cross-Layer Optimizations in Storage: A Workflow-Optimized Storage System," *Submitted to FAST*, 2013.
- [20] D. S. Katz, T. G. Armstrong, Z. Zhang, M. Wilde, and J. M. Wozniak, "Many-Task Computing and Blue Waters," *Technical Report CI-TR-13-0911. Computation Institute, University of Chicago & Argonne National Laboratory. arXiv:1202.3943v1*. 2012.

- [21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 59–72.
- [22] D. G. Murray and S. Hand, “Scripting the cloud with skywriting,” in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, p. 12.
- [23] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, “CIEL: a universal execution engine for distributed data-flow computing,” in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011, p. 9.
- [24] M. Hategan, J. Wozniak, and K. Maheshwari, “Coasters: Uniform Resource Provisioning and Access for Clouds and Grids,” in *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, 2011, pp. 114–121.
- [25] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu, “Early evaluation of IBM BlueGene/P,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 23:1–23:12.
- [26] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii, “Accelerating I/O Forwarding in IBM Blue Gene/P Systems,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–10.
- [27] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, M. Wilde, and I. Foster, “Design and Analysis of Data Management in Scalable Parallel Scripting,” in *Supercomputing*, 2012.
- [28] F. Schmuck and R. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters,” in *Proceedings of the 2002 Conference on File and Storage Technologies FAST*, 2002, no. January, pp. 231–244.
- [29] “Lustre website ,” vol. 2009. .

- [30] C. A. Thekkath, T. Mann, and E. K. Lee, “Frangipani: a scalable distributed file system,” *SIGOPS Oper Syst Rev*, vol. 31, no. 5, pp. 224–237, 1997.
- [31] “POSIX.”
- [32] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 29, 2003.
- [33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies MSST*, pp. 1–10, 2010.
- [34] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, and M. Seaman, “GPFS-SNC: An enterprise storage framework for virtual-machine clouds,” *IBM Journal of Research and Development*, 2011.
- [35] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
- [36] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, “Explicit Control in a Batch-Aware Distributed File System,” *SciencesNew York*, pp. 27–27, 2004.
- [37] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels, “Dynamo: Amazon’s Highly Available Key-value Store,” in *SOSP07*, 2007.
- [38] D. Fetterly, M. Haridasan, M. Isard, and S. Sundararaman, “TidyFS: a simple and small distributed file system,” in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011, p. 34.
- [39] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, “DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 1–14.

- [40] 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.
- [41] M. Abd-El-Malek, W. V Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie, "Ursa Minor: Versatile Cluster-based Storage," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies FAST*, 2005, pp. 59–72.
- [42] J. V Huber Jr., A. A. Chien, C. L. Elford, D. S. Blumenthal, and D. A. Reed, "PPFS: a high performance portable parallel file system," in *Proceedings of the 9th international conference on Supercomputing*, 1995, pp. 385–394.
- [43] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight task executiON framework," in *SuperComputing*, 2007.
- [44] I. Raicu, I. Foster, Y. Zhao, P. Little, C. Moretti, A. Chaudhary, and D. Thain, "The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems," *International symposium on High Performance Distributed Computing (HPDC)*. 2009.
- [45] Z. Zhang, D. Katz, M. Ripeanu, M. Wilde, and I. Foster, "AME: An Anyscale Many-Task Computing Engine," in *Workshop on Workflows in Support of Large-Scale Science*, 2011.
- [46] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crumme, D. Reed, L. Torczon, and R. Wolski, "The GrADS Project: Software Support for High-Level Grid Application Development," *International Journal of High Performance Computing Applications*, vol. 15, no. 4, pp. 327–344, 2001.
- [47] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow Management in Condor," *Workflows for e-Science*, 2007.
- [48] A. Hori, Y. Kamoshida, H. Matsuba, K. Ohta, T. Yasui, S. Sumimoto, and Y. Ishikawa, "On-demand file staging system for Linux clusters," in

Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on, 2009, pp. 1–10.

- [49] “FUSE: Filesystem in Userspace.” .
- [50] P. H. Carns, W. L. III, R. B. Ross, and R. Thakur, “PVFS: A parallel file system for Linux clusters,” *of the 4th annual Linux*, vol. 2000, no. October, pp. 317–327, 2000.
- [51] I. Raicu, Y. Zhao, I. Foster, and A. Szalay, “Accelerating Large-scale Data Exploration through Data Diffusion,” *International Workshop on Data-Aware Distributed Computing*. 2008.
- [52] A. Chervenak, E. Deelman, M. Livny, M.-H. Su, R. Schuler, S. Bharathi, G. Mehta, and K. Vahi, “Data placement for scientific applications in distributed environments ,” *IEEE/ACM International Conference on Grid Computing*. 2007.
- [53] T. Shibata, S. Choi, and K. Taura, “File-access patterns of data-intensive workflow applications and their implications to distributed filesystems,” *International Symposium on High Performance Distributed Computing (HPDC)*. 2010.
- [54] U. Yildiz, A. Guabtni, and A. H. H. Ngu, “Towards scientific workflow patterns,” *Workshop on Workflows in Support of Large-Scale Science*. 2009.
- [55] G. A. Alvarez, E. Borowsky, S. Go, A. Veitch, and J. Wilkes, “Minerva: An automated resource provisioning tool for large-scale storage systems,” *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 4, 2001.
- [56] B. Hall, *Beej’s Guide to Network Programming*. Jorgensen Publishing, 2011.

Appendices

Appendix A : MosaStore Functional and Design Specification

Main contributors to this document: Samer Al-Kiswany, Lauro Costa, Matei Ripeanu, Emalayan Vairavanathan.

Other contributors to the MosaStore project and to this document: Gabriel Bezerra, Abmar Barros, Abdullah Gharaibeh, Elizeu Santos-Neto, Thiago Silva, Sudharshan Vazhkudai

1 Objective

This document serves three purposes: First, it serves as an introduction to the MosaStore project. Second, it serves as a reference for the core MosaStore storage system functionality, architecture, and implementation. Finally, it will serve as a roadmap for planned MosaStore extensions.

*MosaStore*¹ is an experimental storage system. MosaStore is designed to harness unused storage space from network-connected machines to build a high-performance, yet low-cost datastore that can be easily configured for optimal performance in different environments, and for different application data access patterns. MosaStore aggregates distributed storage resources: storage space (based on spinning disks, SSDs, or memory) as well as the I/O throughput and the reliability of the participating machines.

We have *two strategic goals with MosaStore*: On the one side, MosaStore is meant to support fully fledged applications that are deployed in contexts where aggregating resources into a specialized storage system is advantageous. For example MosaStore can be used to support many-task applications [1-3], to provide a specialized, high-performance scratch space [4], to support check-pointing [5], or as a glide-in data store [6], [7].

On the other side, we will use MosaStore as a platform to explore and evaluate innovations in the storage-system architecture, design, and implementation. For example, we plan to extend MosaStore to explore the feasibility of cross-layer communication through custom file metadata [8][9][10], explore solutions to automate storage-system configuration [11], to explore support for data deduplication [12], to explore the feasibility of a versatile storage system [13], [14], or to explore techniques to evaluate and minimize the energy footprint of the storage system [15]. These are only a few of the advanced research projects that will exploit (and hopefully contribute) to the MosaStore code base.

The rest of this document is organized as follows: first it briefly presents a number of intended usage scenarios (Section 2), the requirements the MosaStore storage system aims to meet (Section 3), the functionality currently offered (Section 4), the current architecture (Section 5) and

implementation details (Section 6). Section 7 summarizes gaps in the current system implementation and serves as a development roadmap of MosaStore. Finally, the appendixes provide links to other documents and details about installation and MosaStore internals.

2 Background and Usage Scenarios

2.1. Background

Modern large-scale scientific applications span across thousands of compute nodes and have complex communication patterns and massive storage requirements [16]. They aggregate thousands of computing nodes to get ample computational power and storage space. For these applications, the storage system's throughput and scalability plays a key role in the overall application performance.

Moreover, these applications exhibit highly heterogeneous storage requirements over multiple axes such as read vs. write workload composition, throughput, durability, response time, consistency and reliability guarantees. A typical distributed data store will struggle to efficiently satisfy the storage requirements of all these applications as application-specific storage system optimizations are difficult to design, configure, and deploy and, ultimately, may be conflicting among various applications.

Distributed storage systems have been an active research topic for many years. The overarching goals have been to provide scalable, reliable, secure, and transparent storage. Recent designs to support modern scientific applications include: Ceph [17], [18]; GPFS [19], Lustre [20], PVFS [21], Frangipani [22].

One trend to support scalability and performance is to enable high configurability (or versatility) of the storage system to support specialization for specific deployment environments and workloads. For example, Ursa Minor [13] is a versatile storage system which provides multiple data encoding schemes to exploit the tradeoffs between performance and fault tolerance in the context of various workloads.

A second trend is specializing the storage system for a specific workload or deployment environment. For example, Google file system [23] optimizes for large datasets and appending access; BAD-FS [6] optimizes for batch job submission patterns over wide area network connections, and Amazon Dynamo [24] optimizes for intensive objects put/get operations. The aforementioned

¹ Available at: <http://netsyslab.ece.ubc.ca/wiki/index.php/MosaStore>

storage systems often optimize for one specific workload, provide limited configurability, and provide a non-standard API, requiring modifications to the application.

MosaStore differs from these systems in its design and deployment goals. It aims to incorporate a broad set of optimization techniques, enable high configurability at deployment and/or run time, and support multiple applications through customized, per application deployment, all while still providing a standard POSIX API.

2.2. Usage Scenarios

MosaStore is designed to support two main deployment scenarios:

- *Limited-life deployments* (a.k.a., storage system glide-in [6], [7]). In this case, the storage-system is submitted together with a batch application, instantiated on all (or a subset of) the nodes allocated to the application, and has a lifetime coupled with the application lifetime.
- *Long-term storage system* that aggregates node-local storage resources – for deployments in clusters or networks of workstations.

Two of the limited-life deployment scenarios we envision (and already supported) for MosaStore are presented below. We plan to add more scenarios as we explore them.

2.2.1 High-performance Scratch Space for Many-task Applications

The workflow-based processing used by a large number of scientific applications [25], [26], is generally composed of three main phases: stage-in input-data from central (and often external to the compute nodes cluster) storage to the compute nodes local storage, multiple computation stages that communicate through intermediate files, and stage-out the final results to the central storage. These three phases impose an intense workload on the storage system.

To reduce the load on the shared, system-wide storage system applications can temporarily deploy and configure MosaStore to aggregate storage resources available on allocated compute nodes (local disk, SSDs, memory) and use the storage system thus created as a high-performance scratch space to achieve better performance and resources utilization during runtime.

The application will have to deploy and configure this storage system at launch time; then import the necessary data. The storage system will then be used throughout the application's life time for input/output purposes (and the application may even be able to pass hints to optimize storage system performance). Finally, the end result of the application will be copied on the shared/central file system for persistence and the MosaStore temporary deployment terminated. MosaStore is optimized fast mounting and rapid data migration to support these requirements.

As many other classes of applications, instances of different many-task scientific applications may differ regarding their data usage characteristics such as throughput, data life-time, read/write balance, data

compressibility, locality and consistency semantics. MosaStore enables each application to configure the storage system to best support its own deployment. This application-oriented tuning allows applications to optimize MosaStore operations for the target workload.

2.2.2 Checkpointing

Long-running applications periodically write large snapshots to the storage system to capture their current state. In the event of a failure, applications recover by rolling-back their execution state to a previously saved checkpoint. The checkpoint operation and the associated data have unique characteristics [27][28]. First, checkpointing is a write-intensive operation. Second, checkpoint data is written once and often read only in case of failure. Finally, consecutive checkpoint images present the application state at consecutive time steps and, depending on a number of factors (e.g., checkpointing technique used, frequency) may have a high level of similarity.

MosaStore uses two optimizations to improve the overall performance of checkpointing applications. First, it reduces the data transfers and storage space usage by detecting similarities between checkpoint images. Second, it absorbs the bursty checkpointing writes, and asynchronously writes the checkpoints to the central storage. (See evaluation by Al-Kiswany et al. [5]).

3 Requirements

To support the above usage scenarios, MosaStore requirements are:

- **Easy to deploy:** The storage system should be easy to deploy and mount as part of an application's start-up script (e.g., to support glide-in deployments [6], [7]). Further, ideally it should be transparently interposed between the application and the system central storage for automatic data pre-fetching or storing persistent data or results.
- **Easy to integrate with applications:** The storage system should offer a POSIX-like API to facilitate access to the aggregated storage space, without requiring changes to applications. Offering additional, high-performance APIs (e.g., HDF5, NetCDF) might be desirable.
- **Versatility and ability to configure:** The storage system should provide several configuration knobs to support configurability for diverse applications. The system should be easy to configure and tune for a specific application workload and deployment environment. This includes ability to control local resource usage, in addition to controlling application-level storage system semantics, such as consistency and data reliability requirements.
- **Efficiently harness allocated resources to offer high performance and scalability:** The storage system should efficiently use the node-local storage and networking resources to provide high performance access to the stored data. We aim to support O(10,000) concurrent clients,

O(1000) donor nodes, O(1M) files (typical file size is between 1kB to 100 GB), O(50K) directories.

- **Offer efficient storage for partially similar data.** The storage system should support optimizations for workloads producing partially similar outputs (e.g., checkpointing workloads) by supporting versioning and content-based addressability.

Other requirements the storage system might support:

- **Tuneable security:** Different deployments or different applications might lead to different security requirements. In the future, we plan to support a tuneable security levels in terms of access control, data integrity, data confidentiality, and accountability. Note that the security mechanism should be compatible with the security infrastructure deployed on existing production systems.

4 Functionality

This section briefly presents the functionality offered by MosaStore.

4.1. POSIX API Coverage

MosaStore supports most of the POSIX file system calls that are frequently used by applications. Table 4 provides a detailed description for the support level planned for each system call and indicates the implementation status for the calls we do plan to support.

System calls we do not plan to support in the future:

To reduce the complexity of the system, MosaStore does not support system calls related to file system locking (*fcntl()*), and special file control (*iocctl()*). Also it does not support system calls related to I/O multiplexing (*select()* and *poll()*). Mounting system calls (*mount()* and *unmount()*) and quotas (*quotactl()*) might be supported in latter releases according to necessity.

4.2. Support for Extended Attributes

MosaStore enables adding custom <key, value> pair attributes to files and directories. These custom attributes can be used for: application specific custom metadata such data provenance information, or to enable cross layer optimizations [9][10][8]. MosaStore implements the Linux kernel extended attribute API [29] as the API to access and modify custom attributes.

4.2.1 Namespace for Extended Attributes - Reserved Names

The following are reserved custom attributes; i.e., these attributes are reserved for cross-layer communication. In some cases these attributes are read-only – i.e., applications are not allowed to set their values). The list below is tentative and intended only to show possibilities:

- **location:** is an custom attribute that exposes the file location in MosaStore (i.e. the list of nodes storing the specific file).
- **replication level:** is a custom attribute through which an application indicates the desired replication level for a

specific file (otherwise, the system-level value is used by default)

- **stripe-width:** a custom attribute to control data placement, that is, on how many nodes the file data is stored (see section 4.4).
- **data-placement:** a custom attribute to indicate the data placement strategies.
- **deduplication-enabled:** a custom attribute to control whether deduplication is enabled for a specific file. (in case MosaStore supports both modes in a single deployment).
- **versioning-enabled:** a custom attribute to control whether versioning is enabled for a specific file (in case MosaStore supports both modes in a single deployment).
- **file closing modes:** custom attributes that enables the application to specify the semantics of the close operation (i.e., optimistic vs. pessimistic)

4.3. Content-addressable Storage

MosaStore supports two data storage schemes: a *partial* content addressable storage (CAS) solution [27], [28], [30] and a traditional storage solution. CAS brings a number of benefits: e.g., smaller storage footprint and higher throughput for workloads where data objects have high content similarity; and an implicit ability to verify stored data integrity. The administrator can enable/disable CAS². If CAS is enabled then the administrator can choose the scheme to detect block boundaries: fixed-block size or content-based block boundaries³, and can specify the parameters for each of these schemes.

The content addressable storage is *partial* in the sense that it only supports detecting content similarity among the multiple versions of the same file (it does not detect content similarities across files).

Main use cases: checkpointing, workloads with high content similarity between successive versions of the same file.

4.3.1 Support for Versioning

Currently MosaStore supports versioning only if it is configured as a content addressable storage. The current (most recent) version of a file can be referred by the original file name and the user is expected to use an explicit file naming scheme (as described below) in order access the previous version of the file.

For example: Suppose a user created a file named Hello.C Then its previous versions will be accessible by using their names as Hello.C_v1, Hello.C_v2 and so on. Hello.C will

² *Implementation status:* the goal is to be able to support enabling/disabling of CAS dynamically.

³ *Implementation status:* code for variable-size blocks and content-based block boundaries working but not yet integrated in the main branch (December 2010)

always refer to the latest version of the file. The user will be able to list all the available versions by using the list (“ls”) command, and can access the previous versions as regular files in a read-only mode⁴.

4.4. Data Placement

Two decisions need to be made when a new file is persistently stored: Which (and how many) storage nodes should be used to write the file to? And, among these storage nodes, how to distribute the data?

To answer these questions currently MosaStore uses two policies (described below at a high level):

- To write a file the destination storage nodes are selected in a round robin fashion to load-balance.
- Once a set of storage nodes is selected *stripped writes* are used to accelerate the writes. The *stripe_width* is a configuration parameter - setting the stripe width to 1 results in writing the entire file on one storage node.

4.5. Replication

MosaStore employs data replication for fault tolerance and performance⁵. The replication level (i.e. the number of replicas maintained per data block) is configurable.

4.6. Optimistic vs. Pessimistic Operation Semantics

MosaStore supports operation semantics that can be characterized as *pessimistic* or *optimistic*. These manifest at multiple levels. For example:

- *Deciding when to return success after a request to replicate one chunk* (optimistic/pessimistic replication). Optimistic chunk replication is declared successful after a request is successfully launched, while pessimistic replication is declared successful only after all replicated data has been accepted at all destination nodes. Writing the first replica of a chunk is always pessimistic.
- *Deciding when a close() call should return to application* (optimistic/pessimistic file close). In a pessimistic configuration, a *close()* system call returns only after all chunks of the file have been successfully stored at the storage nodes (replicated or not depending on the replication level and on the optimistic/pessimistic replication scheme at the chunk level). For optimistic configuration, a *close()* operation returns immediately to the application. The storage system creates a thread to

complete the write operation and committing the final *blockmap* to the manager.

The application developer or administrator is able to specify the type of replication and type of file-close semantics (e.g., via a configuration flag or via tagging).

5 Architecture

The MosaStore prototype consists of a logically centralized manager, multiple donor nodes and multiple clients as shown in Figure 1.

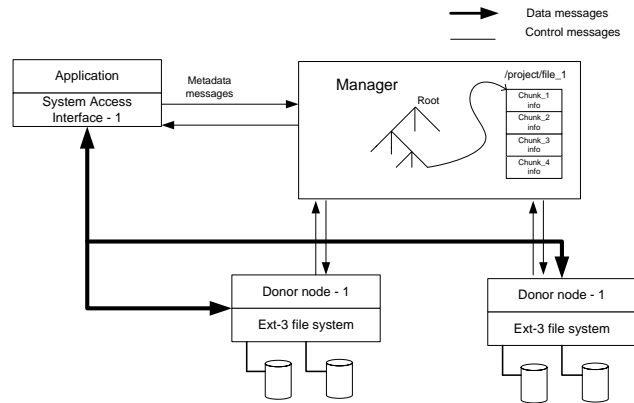


Figure 1: Applications running on the client nodes access the storage system via the system access interface (SAI).

Typically, each of the above three components is deployed on a different node (e.g., a Linux machine) and running as a user-level process. It is also possible to run the donor and the SAI on the same machine.

The following sections present the key design decisions made (Section 5.1) and provide a high-level description of each MosaStore component (the manager §5.2, the donor nodes §5.3, the SAI §5.4), and finally presents the caching design §5.5. Finer-grain implementation details are provided in Section 6.

5.1. Architectural Design Decisions

5.1.1 Stateless Manager

The Manager maintains only the persistent metadata information and does not maintain state regarding the operation in-execution by specific clients (e.g., the manager does not maintain a list of open files and neither it hands off leases). Similarly, the manager does not keep track of the cached data at specific clients. A stateless manager helps limit the system’s complexity and improves the overall system scalability.

5.1.2 Consistency Semantics

For files MosaStore provides session semantics (a.k.a., open-to-close semantics): that is changes in a file will be visible only to clients that open the file after it was closed by the client modifying it. For example suppose a file is opened for reading by application A then, after some time, it is

⁴ *Implementation status:* The code does not currently enforce the read-only access to past versions. A file will be corrupted if an old version is modified. (December 2010)

⁵ *Implementation status:* this is implemented, however, fault-tolerance is not implemented yet— that is, the chunks stored on a failed node are not recreated. (June 2011)

opened by application B for writing (from a different node). Application-A will not be able see the modifications made by application-B. The application developer is expected to be aware about MosaStore's consistency model.

For metadata operations, sequential consistency is provided.

5.1.3 File Chunks

Files are fragmented into chunks that are striped across donor nodes for fast storage and retrieval. Chunks are the addressable data unit of storage. For each file, there is a *chunkmap* mapping the file to its set of chunks and their storage location(s) (i.e., the donor node that stores them).

5.1.3.1 Defining Chunk Boundaries

MosaStore supports two schemes to define chunk boundaries: chunks of fixed size and chunk boundaries based on content⁶. The choice of the chunk boundary scheme is configurable (at present at compile time⁷).

- In the *fixed-size* scheme, the size of the chunk is configured at deployment time and files are divided into equally sized chunks.
- The *content-based* chunking used is similar to that described by [31]. The user can configure the mechanism to use SHA1, MD5 or Rabin fingerprints [32] to detect the chunks boundaries and can provide the other parameters that drive the characteristics of these schemes (e.g., average chunk size).

5.1.3.2 Chunk Naming

Chunk naming in MosaStore can be done in two ways: naming by sequence numbers (to support a traditional storage system) and naming-by-hash (to support content addressability). At present the choice is made at compile time. Specifically:

- *Sequence-based naming*. This scheme generates a unique identifier for each chunk IDs in order to avoid naming conflicts between any two chunks in the system. A combination of IP address of node running the SAI that produced the chunk, the SAI process ID, and uniquely increasing sequence number (the time counter) is used to create the unique identifier (*[IP-Addresses]_[ProcessID]_[SequenceNumber]_[Timestamp]*).

⁶ *Implementation status*: Not yet in the main code trunk (June/2011). Used by StoreGPU project

⁷ *Note*: Dynamic performance tuning functionality may require switching back and forth between these naming schemes in order to measure the performance trade-offs at runtime.

- *Content-based naming*. In this scheme the chunk identifier is the hash value obtained by hashing the chunk content using SHA or MD5 function.

5.1.4 Detecting Chunk Similarity

Similarity is detected based on chunk names (when CAS is enabled⁸): if the content of two chunks does not hash to the same value then chunks are decidedly different. If two chunks hash to the same value they can be either further compared byte-by-byte (as [33] does) or, if one assumes low probability of hash collisions, the two chunks can be directly declared similar (this is the assumption made in the current implementation and is similar to the assumption made in [31]). Currently MosaStore chooses the later and relies on the collision resistant properties of the SHA-1 hash function [34].

5.2. The Metadata Manager

The metadata manager maintains the entire system metadata including: donor nodes' status, file chunk distribution, access control information, and data object attributes. The metadata service is decoupled (to the extent possible) from the data service. That is, clients access the data stored on donor nodes directly and not through the metadata manager.

Since files are divided into chunks and each chunk is identified by its name. It is the role of the metadata manager to maintain the mapping between files and chunks (through a *chunkmap* associated with each file) in addition to the other file metadata.

The Manager uses the NDBM library [35] to persist the metadata. NDBM uses a disk-based extendible hash table to store key-value pairs. The metadata spans across several database tables (the database schema is presented in Appendix 4). All database transactions are serialized using a synchronization mechanism (a single mutex) to provide thread safety.

Figure 2 shows the thread level architecture of the Manager. There are four types of threads:

- *Main thread*. It listens to a port and pre-processes the requests from donor nodes and SAIs.
- *Request processing threads*. These threads are spawned by main thread to serve the requests from SAI and to process the periodic updates and heartbeats from donor nodes.
- *Garbage collector thread*. This thread periodically gets the detail of deleted chunks from NDBM tables and forwards to a donor node for garbage collection.
- *Replication service thread*. Donor nodes failures are monitored by replication thread and durability guarantees are preserved by creating new copies of chunks.

⁸ *Implementation status*: Present only in the SequentialIO module (and not in the GeneralIO) (June/2011)

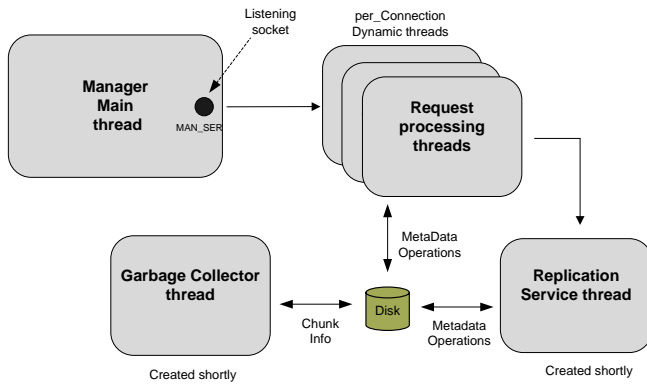


Figure 2: Manager Architecture. The listening socket is shown as a dark circle; the direction of the arrow shows the data flow. The ‘garbage collector’ and the ‘replication’ service threads are spawned at module initialization.

5.3. Donor Nodes

The donor nodes contribute storage space (memory or disk based) to the system. Donor nodes interact with the manager by publishing their status using a soft-state registration process, serve clients’ chunk store/retrieve requests, and participate in the garbage collection mechanism (based on an epidemic protocol).

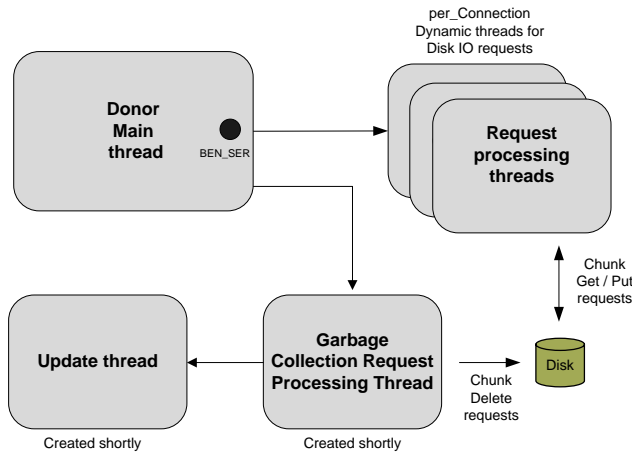


Figure 3: Donor Node Architecture. The listening socket is shown as a dark circle; the direction of the arrow implies that the request processing thread is spawned by the donor main thread. The ‘garbage collection’ and the ‘update’ service threads are spawned at module initialization.

Figure 3 shows the thread level architecture of the donor node. There are four types of threads:

- **Main thread.** The main thread listens to a port for incoming connections. It receives the requests from connection and pre-processes it; garbage collection requests are passed to *garbage collection request processing thread* via transfer queues and disk IO requests are passed to request processing threads.

- **Request processing threads.** Request processing threads are spawned dynamically by main thread for each disk IO requests (e.g: reading / writing chunks).
- **Garbage collection request processing thread.** This thread is created during donor node start-up. Once it receives a garbage collection message, the space is reclaimed and the message is passed to the *update thread* via transfer queues.
- **Update thread.** This thread has two functions: first, it sends periodic updates (heart-beats) to the manager (e.g., containing space left, health status); second, it implements the epidemic protocol that distributes the garbage collection messages to other donor nodes.

5.4. The System Access Interface (SAI)

The SAI (system access interface) is a user-level file system implementation on top of FUSE [36][37](Figure 4).

FUSE introduction: The FUSE infrastructure (Figure 4) consists of two parts: user space FUSE library (libfuse) and a FUSE kernel module. The FUSE library provides a framework through an exported API that is used to implement a user-space file system. The kernel module is composed of *fusefs*, and a character device interface exported through the pseudo device driver */dev/fuse*. The kernel module and libfuse communicate through the character device; file system requests from the virtual file system are received by *fusefs* and transmitted through */dev/fuse* to the library, where they are processed and the results are returned back to *vfs* through the character device to *fusefs*.

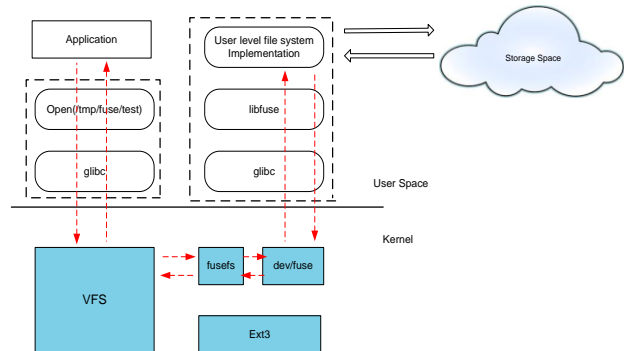


Figure 4: Communication between Application and SAI

The FUSE API [38] defines the interface (callbacks and their arguments) used by application developers to implement FUSE file systems in user space. The FUSE protocol defines the communication between the FUSE kernel module and the FUSE user space library. A detailed description of the FUSE protocol can be found in the FUSE Protocol Specification⁹.

⁹ See: <http://fuse.sourceforge.net/>

Design: MosaStore’s system access interface (SAI) is a collection of implemented FUSE callbacks that provides the mechanism to access the storage space offered by the donor nodes and provides client side optimizations that include caching and similarity detection. The SAI supports the most frequently used POSIX system calls (see section 4.1) and provides easy integration with existing and future applications.

The SAI has two modules, namely the *sequential I/O* module, and the *general I/O* module to serve file system calls. The sequential I/O module is highly optimized for sequential read and writes. The general I/O module serves random reads and writes. Depending on how a file is open, callbacks are dispatched to one of these two modules.

At the thread-level, the SAI contains a permanent thread called *SAI main thread*¹⁰ plus dynamic threads are explicitly spawned by the SAI to process some of the callbacks associated with each of the above modules as a result of the file system calls to serve different tasks in parallel. More specifically:

- [sequential I/O module] *Get Chunk thread* is spawned to fetch chunks while serving a file read request. It terminates after serving the *read()* request.
- [sequential I/O module] *Agent Run thread* is spawned dynamically to send the chunks to donor nodes during a file *write()*.
- [sequential I/O module] *Commit and Free threads* are used to update/send the metadata information to the Manager, once the file is closed after a write.
- [general I/O module] *General IO Send File thread* is used to send the data and metadata of an optimistically opened file via the general IO module.

5.5. Caching

Caching can significantly improve performance. This section discusses caching issues at each of the three MosaStore components. Implementing a caching mechanism at the metadata manager itself is not a priority: The manager already takes advantage of the caching provided by NDBM library. Also implementing another caching layer on top of NDBM might be complex and the potential performance gains are not clear.

Caching at donor nodes might not be necessary if the donor nodes are going to use DRAM based storage (e.g., for diskless nodes such as those of BG/P machine).

Hence, MosaStore provides only client-based caching at the SAI. The current design caches metadata in order to reduce the burden on the manager. The cached metadata includes the list of files in a directory and their attributes.

The cached metadata is invalidated after a configurable time period¹¹.

In the future, different caching and pre-fetching schemes for donor nodes and the metadata manager can be investigated and combined with dynamic performance tuning functionality.

6 Implementation Details

To provide a more detailed presentation of MosaStore internals, this section illustrates the most important workflows currently implemented in MosaStore prototype. The following section (Section 7) provides a detailed discussion on missing and planned additional functionality.

In MosaStore file input/output operations can be performed either using *SequentialIO* module or *GeneralIO* module. *SequentialIO* is highly optimized to support *high-throughput write-once* and *read-many* workloads [16] while *GeneralIO* has few optimizations and designed to support workloads with *append-only* and *random read/write* patterns.

These modules help MosaStore to better support IO-optimizations with less implementation complexity. Currently MosaStore uses the flags passed-in a *file-open* call to identify the suitable IO module. In addition all the write operations to an already existing file are always forwarded to *GeneralIO*. The following sections (6.1 and 6.2) describe the *read-many* and *write-once* workflows of *SequentialIO* and the available special optimizations. The section 6.3 explains *random write* and *append-only* workflows of *GeneralIO* module.

6.1. File Read

The processing steps during the read operations are the same regardless of the naming scheme (by sequence, or by hash) (*Figure 5*).

¹⁰ If FUSE is configured as ‘multi-threaded’ then multiple threads will be spawned by FUSE itself to process these callbacks.

¹¹ *Implementation Status:* This functionality is to be implemented by Emalayan (November 2010)

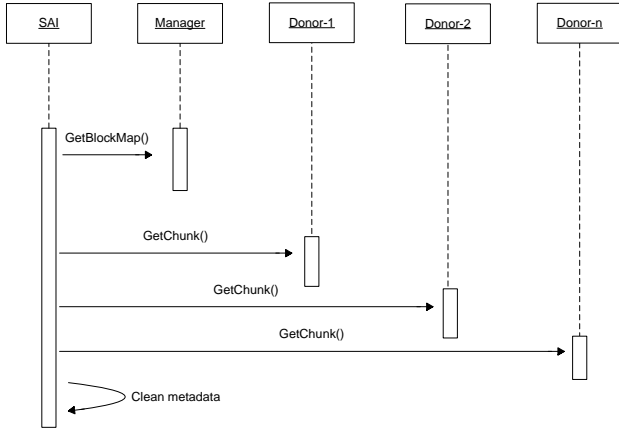


Figure 5: Read workflow for Sequential IO module (full arrows - synchronous communication; half arrow - asynchronous communication; name of the messages are indicated on top of an arrow)

File read workflow

1. Upon a file open request from an application, the SAI sends a request to the manager and downloads the relevant *blockmap* (*blockmap*: metadata information consists of file attributes and the file *chunkmap*, *chunkmap*: contains information about where each chunk is stored).
2. The SAI updates its local *metadata cache*.
3. According to the read request (based on the current reading location), the SAI downloads the corresponding data chunks from donor nodes (*GetChunk()*).
4. Upon a file close request, SAI removes the *blockmap* from the metadata cache.

Note: File *read-open* requests will not go through manager if the *blockmap* exists in the metadata cache.

6.1.1 Read Ahead Optimization

The SAI implements a read-ahead optimization. When a file is opened for read, the SAI spawns a pool of n reading threads that fetch data from donor nodes. The number of threads n is configurable. When the threads are created, they start prefetching the first n chunks of the file to the SAI read buffer. Once a chunk is completely read by the application, one of the threads will fetch a new chunk.

6.2. Writing Files

MosaStore performs a number of optimizations for files opened via *SequentialIO* module. This module only supports new files that are opened in write-only mode. The steps performed during the write operations are slightly different for different naming schemes (by sequence or by hash) and depend on whether naming by content and versioning are enabled. The section starts with presenting the most basic workflow and extends it for various other configurations. The workflow for a sequential write is shown in Figure 6.

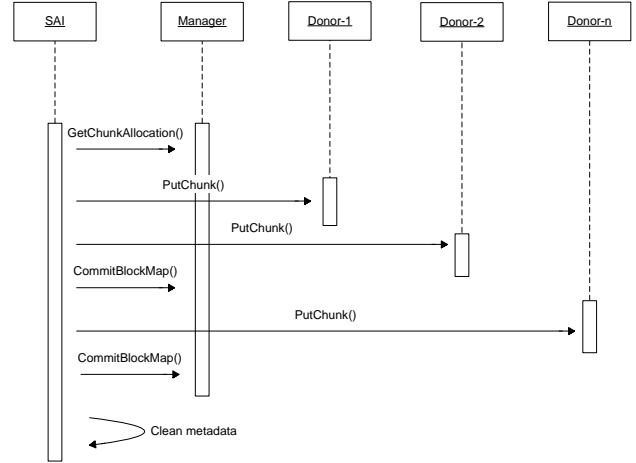


Figure 6: Write workflow for the SequentialIO module

New file, sequential write workflow.

Assumptions: - sequence-based naming, no replication, file versioning is disabled, sequential write

1. When a file is opened for write, SAI gets the *chunkmap* from the manager and instantiates a *blockmap* by combining the *chunkmap* and attributes. The manager creates a *chunkmap* by allocating the chunks in a round-robin fashion from the donor nodes.
2. After opening the file for write, typically, multiple *write()* requests are issued by the application. The data is written into a temporary buffer by the SAI.
3. When the size of the data written in the buffer reaches the size of a complete chunk, the data is sent to a donor node as a new chunk (*PutChunk()*). The new chunk metadata is added to the file *blockmap*.
4. The current *blockmap* is committed at the manager (*CommitBlockMap()*) and the status of commit operation is returned to the application if a *flush()* callback is received from the *libFuse*.
5. Once the file is closed by application, the SAI commits the final *blockmap* at the manager and cleans the metadata cache. The *libFuse* always issues a *flush()* when application closes a file.

Note 1: Using a content-based naming scheme. The write operation with content based scheme is similar to the write operation with sequence based scheme with the following additions. (The step number corresponds to the step number in the workflow above):

2. During the write operation the written data is written to a temporary buffer. When the temporary buffer reaches a defined size the buffer is processed to detect possible chunks boundaries.
- 3.1 If a chunk boundary is detected then a new chunk is written to the donor nodes and the chunk metadata (including the chunk hash value) is added to the file chunk map. The chunk boundary can be detected either based on fixed-size or variable-size. The algorithm to detect variable-size chunk boundaries is described in [39], [40]

3.2 *Else* (a chunk boundary was not detected and the maximum buffer size is reached) an artificial chunk boundary is inserted.

Note 2: Adding file versioning. Currently only in the content based naming scheme MosaStore supports versioning. The write operation with versioning (or with similarity detection) is similar to the write operation presented so far with the following changes (The step number corresponds to the step number in workflows above):

1. When the file is opened for write, the SAI retrieves the file *blockmap* (attributes and *chunkmap*) of the file's latest version from the manager. Then the SAI also creates a new *blockmap* for the new version of the file.
3. When a new chunk is detected (either through path 3.1 or 3.2 of Note 1) and its hash is computed, the SAI will search for the chunk's hash value in the file's latest version *blockmap* before sending the new chunk to the donor node. If a matching hash value is found, the SAI copies the chunk metadata from the latest version to the new version and no data is transferred to donor nodes. If no matching hash value is found, the process continues as before.

Note 3: Adding replication. If replication is enabled for a file¹², chunks will be copied on multiple donor nodes. Replication can be done either pessimistically or optimistically. An application will not be notified regarding the failure of optimistic replication. In this case metadata information of the file will contain some invalid donor node information and the SAI will transparently manage this failure by detecting invalid donor nodes and fetching the data from available donor nodes. The write operation of replicated (optimistic) file is similar to the original write operation with the following changes (Figure 7).

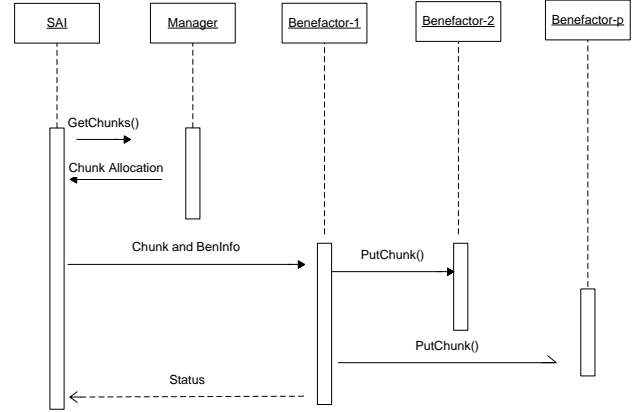


Figure 7: Replication work flow of a file (*replication-semantics* = 2 and *replication-level* = 3)

- 1 When the SAI contacts the manager, it indicates the replication level and the manager replies with a larger space allocation (*chunkmap*).
- 3.1 Then SAI chooses a benefactor from replicas list and forwards the chunk and the list of replicas to it. This benefactor firstly stores the chunk in its local disk and forwards the chunk to the remaining benefactors.
- 3.2 Then, it sends a status message to SAI to indicate whether the operation completed successfully or not.

During pessimistic replication, an application is notified about the status of the replication during a file close operation. Two parameters: *replication-semantics* (used to vary the replication between pessimistic to optimistic) and *replication-level* (maximum number of replicas) are used to configure the replication. The SAI always pessimistically replicates the chunk in some number of replicas equal to *replication-semantics* and then optimistically replicates the chunk the chunk in other replicas ($1 \leq \text{replication-semantics} \leq \text{replication-level}$). Further fault-tolerant service in the manager always monitors the failed donor nodes and replicates the data accordingly.

Note 4: Using pessimistic/optimistic file close. Files can be opened through *SequentialIO* either in optimistic mode or pessimistic mode. In pessimistic mode SAI flushes all the buffered data and metadata to the manager and donor nodes and returns the status of the operation to the application during a file close (6.1 and 6.2 explain the operations in pessimistic mode). A successful close guarantees that the file is safely stored in the system.

In optimistic mode, the SAI immediately return success to the application upon a file close and flushes the stored data and metadata asynchronously. The safety of the file is not guaranteed. All the above mentioned optimizations are supported in both modes.

¹² Currently, the replication level it is configuration parameter for the entire deployment (for all created files). In the current implementation it assigns the replication level defined in the configuration file to all files. In the future the replication level will be per file and communicated through the tagging mechanism (February'11).

6.3. Read/Write

Read/Write operations through *GeneralIO* is different from *read()* only and, *write()* only workflows in *SequentialIO*: here an existing file is downloaded entirely by SAI and both *read()* and *write()* operations occur locally. When a modified file is closed by an application, the file is stored in donor nodes and its metadata updated at the manager. Compared to previous implementations described in Section 6.1 and 6.2 this approach is inefficient because during a file open call entire file is fetched and stored at the client side.

GeneralIO also supports optimistic/pessimistic file close operations, content based naming scheme and replication. In contrast to *SquentialIO*, *GeneralIO* caches both data and metadata to better support applications. The caches are can be flushed either during a file close (pessimistic-close) or after configurable interval (optimistic-close). The read operation of a file is similar to the 6.1 with the following changes.

- 1 SAI checks whether the file is available in the local cache and if the file is not available it download the *blockmap* from the manager.
- 2 The SAI increases the reference-count of the file by 1.
- 3 The SAI downloads all the chunks from the benefactors and creates a local file for read operations. Then SAI updates its local data and metadata caches. All read operations are performed on this local copy.
- 4 Upon a file close request, SAI reduce the reference count and return success to the application. The cached metadata and data is cleaned if reference-count is zero.

Note: If a file is available in local SAI cache all the read operations will be served locally. And if a file is opened with optimistic close mode, SAI cleans both the data and metadata cache after a configurable interval.

The write operation of file is similar to the 6.26.1 with the following changes.

- 1 When a file is opened for write SAI communicate with manager to check whether the file is already available in the system. If the file is not available it creates a new file locally.
- 2 All the writes are written to the local file by SAI.
- 4 Upon a *flush* request, the SAI sends a request to the manager with the current file size and request for new chunk allocation and instantiates a *blockmap*. Then SAI sends the chunks to donor nodes and commit the *blockmap* at the manager.
- 5 Once the file is closed, SAI again do the step-4.

Note: If a file is available in local SAI cache all the write operations will be served locally. And if a file is opened with optimistic close mode, SAI flushes both the data and metadata cache after a configurable interval.

6.4. File Deletion¹³ (*unlink()*)

1. The SAI sends a message to the manager to delete the file.
2. The manager replies back with the status of deletion operation.
3. SAI removes this entry from its local metadata.

6.5. Getting file attributes, reading directory information: *getattr()* and *readdir()*

1. The SAI checks whether file metadata information is available in the local cache. If it is available, the request will be served from local metadata. If it is not available, it will send a message to the manager to download the metadata.
2. The manager replies back with the metadata.
3. SAI updates its local cached metadata and serves the request.

6.6. File Standard Attributes, Extended Attributes and Directory Operations

The workflows for implementing the standard file attribute operations (*chmod()*, *access()*, *mtime()*), extended attribute operations (*setxattr()*, *getxattr()*, *removexattr()*, and *listxattr()*) and directory operations (*rename()*, *rmdir()*, *mkdir()*) are similar. All these functions require communication with the manager only. As a result the workflow is:

1. The SAI sends a request to the manager to update/download metadata.
2. The manager replies back with the status of operation and the requested metadata (in case of *getxattr()/listxattr()*).

6.7. Truncate

Currently the system only supports *truncate()* to zero length files (a common use). For a detailed discussion of what's needed to fully support *truncate* see Appendix 6.

6.8. Space Management or Garbage Collection

This section explains the garbage collection mechanism used by MosaStore to reclaim the space after file deletion.

For garbage collection; one approach is to let the manager to directly communicate with relevant donor nodes to recover the space. This will increase the manager's burden on larger deployments due to the large number of TCP connections and the number of messages to be sent. Another approach is to spread the list of deleted chunks among benefactors in an epidemic fashion. This will reduce the burden on Manager considerably with some challenges in establishing the epidemic protocol.

We have implemented garbage collection using epidemic protocol in MosaStore. The garbage-collection supports the

¹³ *Implementation Status*: Currently (November 2010) MosaStore only removes the file's metadata at the manager. The actual data (chunks) in donor nodes are freed only the file is using sequence-based naming scheme. We are working on a garbage collection scheme.

both naming schemes. Each chunk has a reference-count which tracks the number of files shares the same chunk. Upon a file deletion reference count of the chunks are adjusted and the chunks with reference-count equal to zero are written to the database for deferred garbage collection. Important steps of the epidemic protocol are discussed below:

- **Bootstrapping the epidemic protocol and Spreading membership.**

Each donor nodes maintains the entire list of donor nodes in the system. The manager uses epidemic protocol to spread the information about the newly joined donor nodes to the rest of the donor nodes in the system. So the information about the new donor nodes are added to the manager's pending membership list and then later published with the garbage collection message in an epidemic fashion. After forwarding the membership information the manager cleans its pending memberships list. The manager forwards the list of all the active donor nodes in the system if a donor node joined newly.

- **Spreading garbage collection messages.**

The manager gets the chunks to be garbage collected from the database. Then the manager initiates the epidemic protocol by forwarding the garbage collection message to a randomly selected donor node. After forwarding the message, the manager removes the chunks name from database.

Every donor node which receives a garbage collection message, forwards it to some other randomly selected donor-nodes. The message is spread to the rest of the system in this manner.

- **When to stop the protocol.**

Each garbage collection message has a non negative time-to-live value. The manager calculates this value and forwards it with the garbage collection message. A donor node reduces this value by one before forwarding it to another donor node. If a donor node receives a garbage collection message with time-to-live value equals to zero, then it reclaims the storage space and stops the message spreading; basically the epidemic protocol is stopped.

- **Handling failed donor nodes and spreading failed node information.**

Each donor nodes is responsible to identify the failed donor nodes. If a donor node (node-A) cannot connect to another donor node (node-X) in several tries, then node-X is treated as failed and removed from node-A's membership list.

Modeling of the epidemic protocol.

The following section presents the mathematic model of the epidemic protocol.

fan-out (f) - The number of donor nodes selected at a time to spread a garbage collection message.

time-to-live (l) - Number of hops (donor nodes) that a garbage collection message can pass before termination.

P - Probability of a donor node not getting garbage collection message

N - Total number of nodes in the system

Total messages in the system $T = \frac{(f^l - 1)}{(f - 1)}$

Therefore $P = (1 - 1/N)^T \Rightarrow T = \frac{\log P}{\log (1 - \frac{1}{N})}$

$$\frac{(f^l - 1)}{(f - 1)} = \frac{\log P}{\log (1 - \frac{1}{N})} \quad \text{- Equation-1}$$

In MosaStore P is kept as a constant. The *time-to-live* value is calculated by the manager for a particular *fan-out* to satisfy probabilistic guarantee P using the Equation-1. MosaStore provides two configurable parameters namely “*fan-out*” and “*garbage-collection-interval*” to control the epidemic protocol. “*garbage-collection-interval*” determines the time period between two garbage collection messages.

Epidemic protocol and manager failure:

During a manager restart all the benefactors in the system will connect back, so the epidemic protocol can be established as explained above.

7 Gap Analysis and Design Decisions

This section discusses the gaps between the functional specification and the current working prototype of MosaStore. The goal of this section is to guide the MosaStore team in the planning (by enumerating the gaps) and development process (by providing detailed discussion of planned functionality).

7.1. POSIX API

Table 4 describes the current implementation status of the POSIX API support in MosaStore and outlines the calls that will be developed. To help with prioritization, the table also presents an estimate for the effort associated with implementing each system call.

7.2. Caching

SAI based caches should be invalidated after some configurable period.

7.3. Support for glide-in usage scenario

Since, in this scenario, MosaStore is deployed at the start-up of an application and used until application finishes, it is necessary to provide an efficient way to import and export data from/to other file systems (e.g., GPFS / Lustre). Here the challenges are that staging data in/out should be (ideally be) transparent to the application (or at least simple) while providing consistency, high performance and fault tolerance.

7.4. Fault Tolerance

We need to deal with the failures of all three components of the system as well as with network failures (message loss and connectivity loss).

7.4.1 Donor node failures

A donor node may be unreachable due to software, machine, or network failures. In these cases, all the chunks stored in the donor node will be unavailable temporarily or permanently. A donor node will always validate its data if it connects to the file system after a failure. The functionality to be implemented to provide fault tolerance is to restore on other donor nodes the chunks that were lost due to failure¹⁴:

Fault detection: Donor nodes currently send regular heartbeat messages to the manager. With this information, the manager can detect failed donor nodes and mark them as dead (note to deal with temporary failures).

7.4.2 Manager Failures

The manager may fail and lead to an entire storage system malfunction due to software crashes, machine failures and network partition/failures. Current design will not address these problems and this task is scheduled for the next phase of the project that focuses on a complete metadata manager redesign to provide scalability and fault tolerance.

7.4.3 Client node failures

Need to argue that they do not leave in the wrong state.

7.5. Security

The security is currently relaxed in MosaStore since one of the main use cases is as an application-dedicated storage system and, generally, will be used in trusted network environments. We have explored designs that include configurable security levels in [ref]

7.6. Inter-file similarity detection

The current system does not support similarity detection across files (i.e., it supports only similarity detection across multiple versions of the same file.). We will carefully evaluate the potential gains from supporting this feature (as supporting it involves a complete metadata redesign)

7.7. Other areas for improvement

There are multiple additional areas to improve. Firstly, MosaStore extensively uses blocking socket calls for sending and receiving operations. Second, MosaStore creates one thread per open connection. Third, MosaStore uses temporary TCP connections to communicate among the manager, donor nodes and SAIs – reusing connections is feasible to improve performance. A better approach is to use

non-blocking or asynchronous I/O calls for socket I/Os with persistent TCP connections.

7.8. Known Implementation Shortcuts

This appendix lists known implementation shortcuts:

- `truncate()` only supports `truncate(path, 0)`. Here the file is deleted and a new file is created.
- `chmod()` operation is ignored and all files have a `0x777` access mode
- `chown()` operation is ignored and all users can access all files
- `utimens()` operation is ignored. The SAI returns `modification_time` and `access_time`
- One useful exercise is to try to find ‘unsafe’ library calls in the code (e.g., calls to routines that are not thread safe) and make sure there are no concurrency problems.
- Issue with updates vs. insert/delete – leads to coarse granularity in the metadata and to wired interaction with garbage collection

8 References

- [1] J. Yu and R. Buyya, “A taxonomy of scientific workflow systems for grid computing,” *ACM SIGMOD Record*, vol. 34, no. 3, p. 44, 2005.
- [2] I. Raicu et al., “Towards Loosely-Coupled Programming on Petascale Systems,” *IEEEACM Supercomputing 2008*, 2008.
- [3] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, and M. Wilde, “Design and Evaluation of a Collective IO Model for Loosely Coupled Petascale Programming,” *Science*, 2008.
- [4] H. M. Monti, A. R. Butt, V. Tech, and S. S. Vazhkudai, “/ Scratch as a Cache : Rethinking HPC Center Scratch Storage,” *ICS*, 2009.
- [5] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, “stdchk: A Checkpoint Storage System for Desktop Grid Computing,” *2008 The 28th International Conference on Distributed*

¹⁴ Implementation status (June/2011). Partially implemented - we had a 496 project on this

- Computing Systems*, vol. 0, pp. 613-624, 2008.
- [6] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Explicit Control in a Batch-Aware Distributed File System," *SciencesNew York*, pp. 27-27, 2004.
 - [7] A. Hori et al., "On-demand file staging system for Linux clusters," *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1-10, 2009.
 - [8] E. Santos-Neto, S. Al-Kiswany, N. Andrade, S. Gopalakrishnan, and M. Ripeanu, "Enabling Cross-Layer Optimizations in Storage Systems With Custom Metadata," *Proceedings of the 17th international symposium on High performance distributed computing - HPDC*, p. 213, 2008.
 - [9] S. Narayan and J. A. Chandy, "ATTEST: ATtributes-based Extendable STorage," *Journal of Systems and Software*, vol. 83, no. 4, pp. 548-556, 2010.
 - [10] J. Stribling et al., "Flexible, wide-area storage for distributed systems with WheelFS," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009, pp. 43-58.
 - [11] L. B. Costa and M. Ripeanu, "Towards Automating the Configuration of a Distributed Storage System," 2010.
 - [12] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li, and V. Inc, "Decentralized Deduplication in SAN Cluster File Systems." .
 - [13] M. Abd-El-Malek et al., "Ursa Minor: Versatile Cluster-based Storage," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies FAST*, 2005, pp. 59-72.
 - [14] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu, "The case for a versatile storage system," *SIGOPS Oper Syst Rev*, vol. 44, no. 1, p. 10, 2010.
 - [15] J. Kim, J. Chou, and D. Rotem, "Energy Proportionality and Performance in Data Parallel Computing Clusters," *SSDBM*, 2011.
 - [16] J. Wozniak and M. Wilde, "Case studies in storage access by loosely coupled petascale applications," *Workshop on Petascale Data Storage*, p. 16, 2009.
 - [17] S. Sinnamohideen, R. R. Sambasivan, and J. Hendricks, "A Transparently-Scalable Metadata Service for the Ursa Minor Storage System," *Most*, 2010.
 - [18] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-Scale File Systems," *2008 The 28th International Conference on Distributed Computing Systems*, pp. 403-410, 2008.
 - [19] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 2002 Conference on File and Storage Technologies FAST*, 2002, no. January, pp. 231-244.
 - [20] P. J. Braam, "The Lustre Storage Architecture," *White Paper Cluster File Systems Inc Oct*, vol. 23, 2003.
 - [21] P. H. Carns, W. L. III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," *of the 4th annual Linux*, vol. 2000, no. October, pp. 317-327, 2000.

- [22] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: a scalable distributed file system," *SIGOPS Oper Syst Rev*, vol. 31, no. 5, pp. 224-237, 1997.
- [23] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 29, 2003.
- [24] G. DeCandia et al., "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205-220, 2007.
- [25] T. Oinn et al., *Workflows for e-Science*. Springer London, 2007, pp. 300 - 319.
- [26] I. Raicu et al., "Towards Loosely-Coupled Programming on Petascale Systems," *IEEEACM Supercomputing 2008*, 2008.
- [27] A. Liguori and E. Van Hensbergen, "Experiences with Content Addressable Storage and Virtual Disks," in *Proceedings of the Workshop on IO Virtualization WIOV*, 2008, vol. m.
- [28] P. Nath et al., "Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines," in *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006, pp. 71-84.
- [29] R. Love, "Chapter 12. The Virtual Filesystem," in *Linux Kernel Development Second Edition*, Sams Publishing, p. 432.
- [30] Z. Youhui and W. Dongsheng, "Applying File Information to Block-Level Content Addressable Storage," *Tsinghua Science and Technology*, vol. 14, no. 1, pp. 41-49, 2009.
- [31] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, p. 174, 2001.
- [32] M. O. Rabin, *Fingerprinting by random polynomials*, no. TR-15-81. 1981, pp. 15-18.
- [33] S. Rhea, R. Cox, and A. Pesterev, "Fast, Inexpensive Content-Addressed Storage in Foundation," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008, pp. 143-156.
- [34] D. Eastlake and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)," no. 4634. IETF, 2006.
- [35] Berkeley, "New Database Manager (NDBM) library," 1986.
- [36] "FUSE: Filesystem in Userspace." .
- [37] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, p. 206, 2010.
- [38] I. Sun Microsystems, "FUSE Kernel Operations Function Specifications," 2006.
- [39] A. Z. Broder, "Some applications of Rabin's fingerprinting method," *Sequences II Methods in Communications Security and Computer Science*, pp. 143-152, 1993.
- [40] A. Tridgell and P. Mackerras, "The rsync algorithm," *Imagine*, no. TR-CS-96-05. 1996.

Appendix 1: RELATED LINKS

MosaStore website: <http://mosastore.net> or <http://netsyslab.ece.ubc.ca/wiki/index.php/MosaStore>

Appendix 2: Installation instructions

Download the MosaStore tar-ball and extract it in a local directory. MosaStore is composed of three main modules namely manager, benefactor and file system interface and the extracted MosaStore source code directory contains different directories for each module. Each module can be separately built using the Makefiles and further direction is provided in the “readme.txt” files available with each modules. Manager, Benefactor and SAI binaries should be started as mentioned below to deploy the MosaStore file system.

Manager:

Run the binary using “./manager [port_number] [config_file]” where;

[port_number]: is the port number through which the manager accepts incoming connections.

[config_file]: is the path to the configuration file to be used.

Benefactor:

Run the binary using “./benefactor [config_file]” where;

[config_file]: is the path to the configuration file to be used.

SAI:

Run the binary using “./mosastore -o direct_io -o sync_read -o allow_other -c [config_file] [mount_point]” where ;

[mount_point]: is the mounting point (e.g. /tmp/mntpoint) and make sure that the mounting point is empty.

[config_file]: is the path to the configuration file to be used.

Note: Always [config_file] parameter is optional and if it is not provided the MosaStore File System will check the current directory for configuration file. If configuration file is not available in the current directory then binaries will terminate.

Appendix 3: Configuration instructions

This section presents the set of configuration options at each of the main three components of the system: the manager, the donor nodes, and the SAI. (Default values are provided together with the default configuration files used by MosaStore).

Table 1: Manager Configurations

Configuration	Descriptions
stripe_width	This will determine the number of benefactors that the client will strip the data during file write operations.
chunk_size	The maximum size of a chunk.
max_num_ben	The maximum possible number of benefactors in the system
log_mode	Log mode could be : DEBUG, VERBOSE, ERROR, FATAL, RESULT, OFF (Optional)
log_file	Name of the log file (Optional). If log_file name is not specified and log_mode is not OFF, all the messages will be written to standard output.

Table 2: Benefactor configurations

Configuration	Descriptions
manager_name	The hostname or the IP address of the manager.
manager_port	The manager port number.
benefactor_path	The path to the local directory where the benefactor stores the chunks.

aggregation_type	Aggregation type which specifies the benefactor's chunk storage medium, this could be <DISK> or <MEMORY>.
disk_space_size	The donated disk space size in MB.
memory_space_size	The donated memory space size in MB, this is effective if aggregation type= MEMORY
update_period	Determines the time interval between two successive status update messages. Status update messages are used to inform the status of the benefactor to the manager.
benefactor_address	The local IP address the benefactor which should be used while creating listening sockets. (Optional – this is useful if a machine has multiple IP addresses).
log_mode	Log mode could be : DEBUG, VERBOSE, ERROR, FATAL, OFF (Optional)
log_file	Name of the log file. If not specified and the log_mode is not OFF, the log messages will be sent to standard output.

Table 3: SAI configurations

Configuration	Descriptions
manager_name	The hostname or the IP address of the manager
manager_port	The manager port number.
chunk_naming	Naming scheme can be either SEQNUM or HASH and selected according to the workloads. For example high similarity workloads need HASH naming scheme to enable similarity detection.
commit_scheme	Commit scheme might be NOOVERWRITE or OVERWRITE or VERSIONING. NOOVERWRITE: writing a new file with the same name as an existing file will fail; OVERWRITE: system allows to create new file with an existing file name; VERSIONING: system stores
num_reserve_chunks	Specifies the capacity of chunk repository. Higher capacity will reduce the communication between SAI and the manager regarding chunk allocations.
write_interface_type	Used to select the write interface type. SLIDINGWINDOWWRITE: Sliding window write interface, is the typical setting.
memory_size	The memory space allocated for the buffers in the write operations, in MB, effects Sliding window interface only.
local_write_directory	Specifies the path to the directory which is used by SAI to store all the temporary files.
fbr_request_buffer_size	Specifies the number of chunks allocated during a read request.
num_threads_per_agent	Specifies the number of threads per write agent (there is an agent per benefactor and at least this value should be one).
cache_update_period	Cache update period in seconds, if this value is set to 0 then the cache is disabled– Samer pls verify ¹⁵
log_mode	Log mode could be : DEBUG, VERBOSE, ERROR, FATAL, RESULT, OFF (Optional)
log_file	Name of the log file

Appendix 4: ER Diagram for Metadata

MosaStore database schema consists of several normalized database tables which hold information regarding files and directories (Figure 8).

TABLE_FILE_IDS and TABLE_DIR_IDS map all the files and, respectively, directories in the file system to unique integers called *file Id* and *directory Id* respectively.

The *directory Id* is used as foreign key to access a directory's attributes, extended attributes, list of files and list of sub-directories which are stored in TABLE_DIR_ATTRS, TABLE_DIR_FILES, TABLE_DIR_XATTRS, and TABLE_DIR_SUBDIRS respectively.

Similarly, the *file Id* is used as foreign key to access a files' attributes, extended attributes, chunk list, and benefactor list which are stored in TABLE_FILE_ATTRS, TABLE_FILE_XATTRS, TABLE_FILE_CHUNKS and TABLE_FILE_BENEFACTORS.

¹⁵ This is not implemented yet. This is what we plan to implement.(February'11)

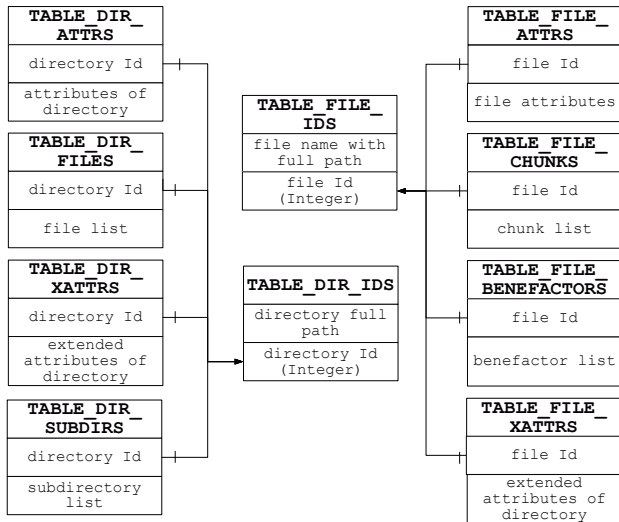


Figure 8: Entity-Relationship diagram of MosaStore Metadata

Appendix 5: POSIX system calls

POSIX call	Current Status	Implementation Effort	Additional Note
close()	Yes		
unlink()	Yes		
chdir()	Yes		
lseek()	Yes		
fstat()	Yes		
rename()	Yes		
mkdir()	Yes		
rmdir()	Yes		
chroot()	Yes		
stat()	Yes		
lstat()	Yes		
dup()	Yes		
dup2()	Yes		
getdents()	Yes		
fchdir()	Yes		
sysfs()	Yes		
read()	Partially	Moderate	Random reads are not supported ()
write()	Partially	Moderate	Random writes and file append is not supported
open()	Partially	High	Wide variety of file flags are not supported
creat()	Partially		See open()
mknod()	Partially	High	Some flags are not supported
readv()	Partially	Low	Dependency with read() function
writev()	Partially	Low	Dependency with write() function
pread()	Partially		Dependency with read() function
pwrite()	Partially		Dependency with write() function
link()	No	Low	
chmod()	No	Low	
fchmod()	No		See chmod()
lchown()	No	Low	
fchown()	No		See lchown()
chown()	No		See lchown()
utime()	No	Low	
access()	No	Low	
sync()	No	Medium	
symlink()	No	Low	
readlink()	No	Low	

truncate()	No	Medium	See Appendix 6.1
ftruncate()	No		See truncate()
statfs()	No	Medium	
fstatfs()	No		See fstatfs()
fdatasync()	No		
fsync()	No		

Table 4: Status of file system calls in current MosaStore implementation

Appendix 6: Design Discussion

This appendix discusses the design issues for some parts of MosaStore and details the shortcuts taken in the implementation.

A. 6.1. truncate() design

API Details: *truncate(file, new_size)* will change the file size to *new_size*.

If *new_size* < *current_size*

then the extra data will be deleted and the file will have its size equal to *new_size*

else the file will be padded with zeros until the file size is equal to *new_size*

Often *truncate()* is called to delete the file content i.e., *truncate(file, 0)*.

Why a complete support of *truncate* needs elaborate implementation? The implementation is complex since *truncate* may have some complex special case (especially with *name_by_hash*):

- if the *new_size* < *current_size* and the *new_size* is not on the chunk boundary here we need to delete the extra chunks, and also to copy the last chunk and create a truncated version of the last chunk, hash the new chunk and update the file blockmap (this is needed to support *truncate* for content addressability) - (this involves new operations on the benefactor to compute the hash and update the file blockmap)
- if *new_size* > *current_size* we need to pad the file with chunks filled with zeros. this involves developing new mechanism to create chunks filled with zeros of varying sizes and computing their hashes and update the file blockmap.
- It is complex to support *truncate* for open files

Possible optimization:

One optimization is to create a special chunk (*chunk_0*) which is not stored on any benefactor.

Adding this special chunk needs special processing in many places in the code:

- read operation to create a buffer and fill it with zeros when *chunk_0* is read
- need to provide special implementation to support modifying existing files with *chunk_0*.
- Garbage collection and replication to ignore this chunk

Possible shortcuts:

- Support *truncate* only when naming chunks by sequence number. this will cut 25% of the implementation but still we need to implement most of the mechanisms.
- Larger shortcut: only support the most common *truncate* operation (*truncate(file, 0)*) here we will delete the previous file and leave it empty. And return error for any *truncate* with *new_size* > 0