AnalyzeThis: An Analysis Workflow-Aware Storage System

Hyogi Sim

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Ali R. Butt, Chair
Changhee Jung
Sudharshan S. Vazhkudai

December 17, 2014
Blacksburg, Virginia

Keywords: Distributed System, File System
Copyright ©2014, Hyogi Sim

AnalyzeThis: An Analysis Workflow-Aware Storage System

Hyogi Sim

(ABSTRACT)

Supercomputing application simulations on hundreds of thousands of cores produce vast amounts of data that need to be analyzed on smaller-scale clusters to glean insights. The process is referred to as an *end-to-end workflow*. Extant workflow systems are stymied by the *storage wall*, resulting from both the disk-based parallel file system (PFS) failing to keep pace with the compute and memory subsystems as well as the inefficiencies in end-to-end workflow processing. In the post-petaflop era, supercomputers are provisioned with flash devices, as an intermediary between compute nodes and the PFS, enabling novel paradigms not just for expediting I/O, but also for the *in-situ* analysis of the simulation output data on the flash device. An array of such *active flash elements* allows us to fundamentally rethink the way data analysis workflows interact with storage systems. By blending the flash storage array and data analysis together in a seamless fashion, we create an analysis workflow-aware storage system, *AnalyzeThis*. Our guiding principle is that analysis-awareness be deeply ingrained in each and every layer of the storage system—*active flash fabric, analysis object abstraction layer, scheduling layer within the storage, and an easy-to-use file system interface*—thereby elevating data analyses as *first-class citizens*. Together, these concepts transform AnalyzeThis into a potent analytics-aware appliance.

Contents

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1

2 Related Work 4

3 AnalyzeThis Storage System 6

3.1 Goals 6

3.2 Overview 7

4 Design and Implementation 9

4.1 Analysis Encapsulation 9

4.1.1 Extending OSD Implementation for AFE 11

4.2 Workflow Engine 13

4.2.1 Workflow Description and DAG 14

4.2.2 Scheduling Heuristics 14

4.3 anFS File System Interface 15

4.4 Provenance 18

5 Evaluation 20

5.1 Experimental Setup 20
5.1.1 Hardware Platform 20
5.1.2 Software 21
5.1.3 Scientific Workflows 21
5.2 AnalyzeThis Performance 22
5.2.1 anFS Performance 22
5.2.2 End-to-End Workflow Experiments 23
5.2.3 Impact of Scheduling Heuristics 25
5.2.4 Scaling Experiments 26
5.2.5 Provenance Performance 27

6 Conclusions and Future Works **28**

Bibliography **29**

List of Figures

3.1 *AnalyzeThis* overview. Figure shows analysis-awareness at each and every layer of AnalyzeThis. 7

4.1 *AnalyzeThis* Architecture. 10

4.2 Analysis object abstraction implemented using database engine. CID: task collection id, OID: object id, Attr. offset: an offset in the attribute page. 12

4.3 anFS architecture shows the namespace, workflow, device manager, and exoFS components with the request flow. 16

5.1 AnalyzeThis testbed. 20

5.2 The DAGs representing the workflows. 21

5.3 End-to-end workflow performance of offline (one and four nodes) and AnalyzeThis (Xeon and Atom AFEs). AnalyzeThis used four AFEs and the round-robin scheduling. Both ideal and real PFS bandwidth based results are shown. 23

5.4 Performance of scheduling heuristics with Atom-based AFEs. 25

5.5 AFE scaling experiments for AnalyzeThis. 26

List of Tables

1.1 Simulation output data per compute node for large-scale application runs on the Titan machine [1]. 2

2.1 Comparison with related active storage systems and worklflow-aware systems. '×' means that a system implements the listed feature, whereas '·' implies that the system does not provide the feature. 5

5.1 Workflow input, output and intermediate data size. 22

5.2 Avg. read/write rate for 1 GB dataset. 22

5.3 Response time for five provenance queries. 27

Chapter 1

Introduction

High-performance computing (HPC) applications—in astrophysics, combustion, climate, and fusion—simulate complex physical phenomena (henceforth referred as *simulations*) on supercomputers with hundreds of thousands of cores, and produce vast amounts of data to a central, shared parallel file system (PFS) such as Lustre [2] or GPFS [3]. For example, the 27 petaflops Titan machine [4] (No. 2 on the Top 500 list of supercomputers [5]) runs massively parallel MPI simulations that routinely write tens of terabytes of data to the Lustre-based Spider PFS [6] (Table 1.1). The large-scale simulations are typically followed by a series of data analysis or post-processing tasks that run on smaller-scale clusters in the same HPC center, to glean insights from the simulation output. Example analysis tasks include statistics, reduction, clustering, feature extraction, as well as legacy application routines. Unlike the simulation, the data analyses are neither FLOP-intensive, nor do they require low-latency communication. Consequently, they are run on clusters that are orders of magnitude smaller and less powerful. The simulation plus the data analysis sequence constitutes an *end-to-end workflow*.

Simulations and data analyses are chained together using resource management (PBS [7] and Moab [8]) or workflow management tools (Kepler [9], Pegasus [10] and DagMan [11]), sharing data via the PFS (*Offline data analysis*). Offline analysis incurs a substantial amount of redundant I/O, as analysis has to read the simulation output from the PFS and write the reduced results back to the PFS. For example, a 12-hour run of the S3D combustion simulation [12] on Titan produces about 10 TB of data every hour. Reading back such large data for analysis exacerbates the I/O bandwidth bottleneck that is already acute in HPC centers. Even though extreme-scale PFS provides high "peak" I/O throughput, applications only realize a fraction of this performance due to heavy I/O contention. Titan's Spider PFS offers a peak I/O throughput of 1 TB/s to its 18,688 compute nodes (299,008 cores), at a rate of only 56 MB/s per node. As these machines scale, the I/O subsystem that has traditionally failed to keep pace with the growth in other system components, e.g., compute and memory, causes a fundamental imbalance, creating a *"storage wall."*

To alleviate the I/O bottleneck, HPC systems are beginning to deploy solid-state devices (SSD) locally on the compute nodes, e.g., Tsubame 2.5 [13], Gordon [14], and Catalyst [15] have hun-

Application	Data Output/Node
CHIMERA (Astrophysics)	4400 KB/s
POP (Climate)	17 KB/s
S3D (Combustion)	170 KB/s
GTC (Fusion)	14 KB/s

Table 1.1: Simulation output data per compute node for large-scale application runs on the Titan machine [1].

dreds of terabytes in aggregate SSD-based storage. The lack of mechanical moving parts, coupled with a superior I/O bandwidth and low latency, has made SSDs an attractive choice for node-local storage on large-scale systems. This emerging use of SSD as a node-local storage has the potential to enable novel paradigms, not just for expediting bulk data output [16, 17, 18], but also for data analysis.

SSDs boast an increasing computational capability on the controllers, which has the potential to facilitate *in-situ* data analysis. In this model, the compute nodes write the outputs to their respective SSDs, where the analysis is performed in-situ; only the final, reduced data is written to the PFS, obviating redundant I/O. Recent explorations such as Active Flash [19, 1] for HPC, and iSSD[20] and Smart SSD [21] for enterprise workloads study the performance and energy tradeoffs of conducting analysis on the SSD. *Such an active processing element can form the fabric of a storage system that is workflow-aware*. In this paper, we aim to realize such a system.

An array of compute node-local SSDs, capable of active processing (hereafter referred as "*active flash elements," AFE*), allows us to fundamentally rethink the way data analysis workflows interact with storage systems. Traditionally, storage systems and workflow systems have evolved independently of each other, creating a disconnect between the two. By blending the flash storage array and data analysis together in a seamless fashion, we create an analysis workflow-aware storage system, *AnalyzeThis*. Consider the following simple—yet powerful—analogy from day-to-day desktop computing, which explains our vision for AnalyzeThis. A *smart folder* on modern operating systems allows us to associate a set of rules that will be implemented on files stored in that folder, e.g., convert all postscript files into pdfs or compress (zip) all files in the folder. A similar idea extrapolated to a petascale simulation would be: writing data to an analysis-aware compute node-local storage system automatically *triggers* a sequence of predefined analysis routines to be applied to the data.

Contributions: We propose *AnalyzeThis*, a file system atop an array of AFEs. Our guiding principle is that analysis-awareness be deeply ingrained within each and every layer of the storage system, thereby elevating the data analysis operations as *first-class citizens*. AnalyzeThis realizes workflow-awareness by creating a novel *analysis data object abstraction*, which integrally ties the dataset on the flash device with the analysis sequence to be performed on the dataset, and the lineage of the dataset (Chapter 4.1). We also explore how scheduling, i.e., both data placement and workflow orchestration, can be performed within the storage, in a manner that minimizes unnec-

essary data movement between the AFEs, and optimizes workflow performance (Chapter 4.2). Finally, we design easy-to-use file system interfaces with which the active flash fabric can be exposed to the user (Chapter 4.3). The FUSE-based file system layer enables users to read and write data, submit jobs, track and interact with them via a */proc-like* interface, and pose provenance queries to locate intermediate data (Chapter 4.4). We argue that these concepts bring a fresh perspective to extreme-scale data analysis. Our results with real-world, complex data analysis workflows indicate that AnalyzeThis can expedite end-to-end runtimes and significantly reduce data movement costs.

Chapter 2

Related Work

Migrating computational loads to disks has been explored before [22, 23]. There is a renewed interest in active processing given the recent advances in SSD storage technology [24]. Recent efforts, such as iSSD [20], SmartSSD [21], and Active Flash [1] have demonstrated the feasibility and the potential of processing on the SSD. These early studies lay the foundation for AnalyzeThis. However, we take significant strides further by building a complete workflow-aware storage system, and positioning it as an in-situ processing storage appliance.

The active storage community has leveraged the object storage device (OSD) protocol to enable computation within a storage device. The OSD T10 standard [25, 26, 27] defines a communication protocol between the host and the OSD. Recent efforts leverage the protocol for different purposes, including executing remote kernels [27], security, and QoS [28, 26]. In contrast, we extend the OSD implementation to support entire workflows, and to integrally tie together the data with both the analysis sequence and its lineage.

Table 2.1 provides a comparison between other closely related efforts and AnalyzeThis along different dimensions, e.g., active storage processing, workflow and provenance-awareness, OSD model, file system interface, and in-situ data analysis. While there are approaches that provide solutions targeting a few dimensions, none of them provide a complete solution, satisfying all of the dimensions. For example, some extensions to PVFS [36] and Lustre [37] provide support for active computation in the PFS, but are not workflow-aware, a key trait for efficient analysis execution. Moreover, these efforts propose to conduct analysis on the I/O node's computing element within the PFS and not on the storage device itself. The ADIOS [38] I/O middleware enables in-situ analysis on a subset of staging nodes alongside a simulation; while workflow-aware, it also only uses the computing elements of the staging nodes. Further, dedicating supercomputer nodes for analysis takes away FLOPS from the main simulation. Instead, AnalyzeThis uses the AFEs on the simulation nodes themselves, obviating the need for a separate set of staging nodes for analysis. Enterprise solutions such as IBM Netezza [35] enable provenance tracking and in-situ analysis, but cannot be readily used for scientific workflows because they lack an easy-to-use file system interface and workflow-awareness. Workflow- and provenance-aware systems, such as PASS [29],

Systems	Active Device	Workflow	OSD Model	Provenance	FS Interface	In-Situ
Provenance-awareness						
PASS [29]	·	·	·	×	×	·
LinFS [30]	·	·	·	×	×	·
VDT [31]	·	·	·	×	·	·
Workflow-awareness						
BadFS [32]	·	×	·	·	·	·
WOSS [33]	·	×	·	·	×	·
Kepler [9]	·	×	·	×	·	·
Active Storage						
Active Disks [34]	×	·	·	·	·	×
Active Flash [1]	×	·	·	·	·	×
iSSD [20]	×	·	·	·	·	×
SmartSSD [21]	×	·	×	·	·	×
Analytics Appliance						
IBM Netezza [35]	·	·	·	×	·	×
Active Computation in PFS						
PVFS [36]	·	·	×	·	×	×
Lustre [37]	·	·	×	·	×	×
I/O Middleware						
ADIOS [38]	·	×	·	·	·	×
AnalyzeThis	×	×	×	×	×	×

Table 2.1: Comparison with related active storage systems and worklflow-aware systems. '×' means that a system implements the listed feature, whereas '·' implies that the system does not provide the feature.

LinFS [30], BadFS [32], WOSS [33], and Kepler [9], are not meant for in-situ analysis. The batch-aware distributed file system (BadFS) is relevant as it attempts to orchestrate I/O-intensive batch workloads and data movement on remote systems, by layering a scheduler atop distributed storage and compute systems. In contrast, AnalyzeThis operates on node-local AFEs, scheduling and colocating data and computation therein. In terms of provenance capture, unlike dedicated systems like PASS and LinFS, linege tracking is a natural byproduct in AnalyzeThis, by virtue of executing workflows within the storage. Thus, AnalyzeThis provides a design and implementation that satisfies all of the aforementioned dimensions (Chapter 4).

Chapter 3

AnalyzeThis Storage System

3.1 Goals

In this chapter, we discuss our key design principles.

Analysis-awareness: The main objective of our design is to introduce workflow-aware semantics into the storage system. The disconnect between storage systems and workflow management engines results in sub-optimal performance for end-to-end workflows. There is an urgent need to analyze the data in-situ, on the storage component, where the data already resides.

Reduce Data Movement: At exascale it is expected that the overall cost of data movement will rival that of the computation itself [39]. Thus, we need to rethink the way data analysis is performed on future machines. We need to minimize data movement between the compute nodes of the supercomputer and the PFS, and between the PFS and other data analysis clusters within an HPC center. Further, we should also minimize the data movement across the AFEs within the storage appliance itself.

Capture Lineage: There is a need to track provenance and intermediate data products generated by the analysis steps on the distributed AFEs. The intermediate data can serve as starting points for future workflows.

Easy-to-use File System Interface: The workflow orchestration across the AFEs needs to be masqueraded behind an easy-to-use, familiar interface. Users should be able to easily submit workflow to the storage system, monitor and track them, query the storage system for intermediate data products of interest and discover them.

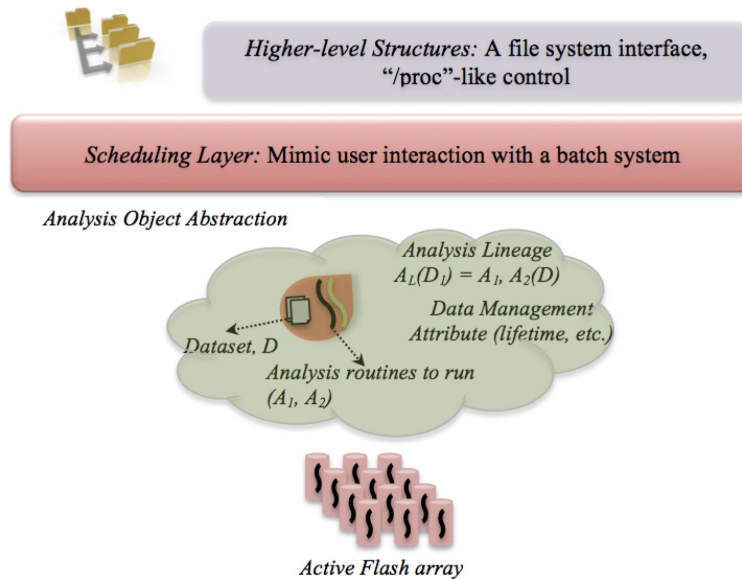


Figure 3.1: *AnalyzeThis* overview. Figure shows analysis-awareness at each and every layer of AnalyzeThis.

3.2 Overview

We envision AnalyzeThis as a smart, analytics pipeline-aware storage system atop an array of AFEs (Figure 3.1). AnalyzeThis can reside on every compute node of the supercomputer or on dedicated "fat" nodes that are provisioned with flash devices. In either case, the system is positioned between the compute nodes and the PFS. Parallel application simulations write their output data to AnalyzeThis instead of the PFS. The output will be analyzed based on the workflow that the user has submitted to the storage system. The final processed data, or any of the intermediate data may be transferred to the PFS, based on lifetime metadata attributes that the user may have associated with the dataset. *Thematic to the design of AnalyzeThis is that analysis-awareness be deeply embedded within each layer of the storage system.* Below is a bottom-up description of the system.

Active Flash Fabric: At the lowest level is the active flash fabric that is capable of running individual analysis kernels. We envision an array of such AFEs that are connected via SATA, PCIe or NVMe.

Analysis Object Abstraction: On top of the active flash array, we propose to create a new data model, the *analysis object abstraction* that encapsulates the data collection, the analysis workflow to be performed on the data, and the lineage of how the data was derived. We argue that such a rich data model makes analysis a *first-class citizen* within the storage system by integrally tying together the data and the processing to be performed (or was performed) on the data.

Workflow Scheduling Layer: The goal of this layer is to mimic how users interact with batch

computing systems and integrate similar semantics into the storage system. Such a strategy would be a concrete step towards bridging the gap between storage and workflow systems. Users typically submit a workflow, e.g., a PBS [7] or a DAGMAN [11] script, to the supercomputer's batch scheduler, which creates a dependency graph and dispatches the tasks onto the compute nodes based on a policy. Similarly, we propose a *Workflow Scheduler* that determines both data placement and scheduling analysis computation in a manner that optimizes both end-to-end workflow performance and data movement costs.

A File System Interface: We tie the above components together into a cohesive system for the user by employing a FUSE-based file system interface with limited functionality (*"anFS"*). anFS supports a namespace, reads and writes to the active flash fabric, directory creation, internal data movement between the AFEs, a *"/proc-like"* infrastructure, and the ability to pose provenance queries to search for intermediate analysis data products. Similar to how */proc* is a control and information center for the OS kernel, presenting runtime system information on memory, mounted devices and hardware, `/mnt/anFS/.analyzethis/`, allows users to submit workflow jobs, track and interact with them, get status information, e.g., load about the AFEs.

Together, these constructs provide a very potent in-situ data analytics-aware storage appliance.

Chapter 4

Design and Implementation

Figure 4.1 presents the architecture of AnalyzeThis. Users submit analysis workflows and write data objects via a FUSE file system interface, *"anFS"* that is mounted on the compute node or host. Thereafter, users can monitor and query the status of the jobs, and search for intermediate data products of branches of the workflow. Central to our design is the seamless integration of workflow scheduling and the file system. anFS provides a file system interface, behind which is a *workflow scheduler* that constructs a directed acyclic graph (DAG) of the workflow. The scheduler produces multiple mini DAGs based on task dependencies and optimization strategies, e.g., to minimize data movement. The mini DAGs comprise of a series of tasks, which the host maintains in a lightweight database. The scheduler dispatches the mini DAGs for execution on the AFEs; AFEs form the bottom-most layer of our system and are capable of running analysis kernels on the device controllers. We use an *analysis object abstraction*, which encapsulates all necessary components of the analysis workflow, including analysis kernels, input and output datasets, and the lineage information of all the objects therein. The analysis kernel is stored as a platform-dependent binary executable object (`.so` format in our implementation), compiled for specific devices as needed, which can run on the AFEs.

4.1 Analysis Encapsulation

We introduce analysis awareness in the flash storage fabric by building on prior work that has demonstrated how to run an analysis kernel on the flash device controller [1, 20, 21]. Our goal, however, is to study how to use such an active storage device, and overlay an *analysis object abstraction* atop. We refer to this as the Active Flash Element (AFE). The construction of an AFE involves interactions with the flash hardware to expose features that higher-level layers can exploit, communication protocol with the host and flash device, and the necessary infrastructure for analysis object semantics. An array of AFEs serve as building blocks for AnalyzeThis.

The first step to this end is to device a richer construct that is much more than just files. Scientific

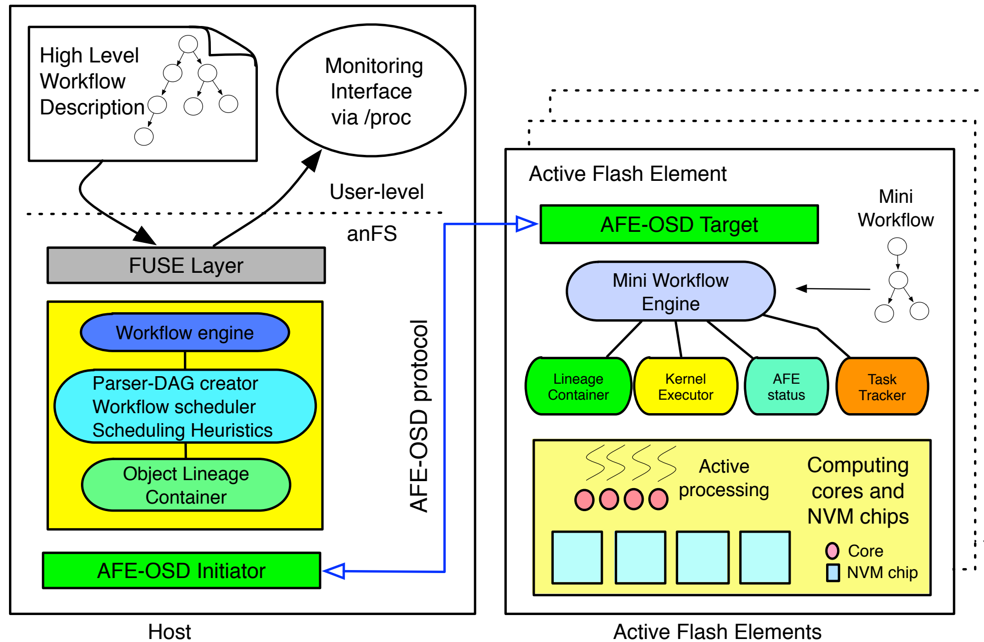


Figure 4.1: *AnalyzeThis* Architecture.

communities have spent a substantial amount of time into deriving a variety of data formats—NetCDF [40, 41], HDF [42, 43, 44, 45], NeXus [46]—that offer many desirable features such as access needs (parallel I/O, random I/O, partial I/O, etc.), portability, the availability of many community tools for processing, efficient storage and self-describing behavior. These are all valuable traits for large scientific data. However, we also need a way to tie these datasets with the analysis lifecycle in order to provide for and support future data-intensive analysis.

To address the above shortcoming, we extend the concept of a basic data storage unit from traditional *file(s)* to an analysis object abstraction that includes a *file(s) PLUS a sequence of analyses that operate on them PLUS the lineage of how the file(s) were derived*. Such an abstraction can be created at a data collection-level (a collection may contain thousands of files, which is more common in several scientific communities, e.g., climate). The analysis data abstraction would at least have either the analysis sequence or the lineage of analysis tools (used to create the data) associated with the dataset during its lifetime on AnalyzeThis. The elegance of integrating data and operations is that one can even use this feature to record *data management* activities as part of the dataset and not just analyses. For example, we could potentially annotate the dataset with a *lifetime* attribute that tells AnalyzeThis which datasets to retain and for how long, e.g., retain only every tenth checkpoint data of a simulation. *What the analysis, lineage and management metadata indicate is that the dataset is now part of a really powerful construct; it is now an encapsulation for something larger than merely a pointer to a byte stream; it is now an active entity that lends itself to on-the-fly analytics.*

4.1.1 Extending OSD Implementation for AFE

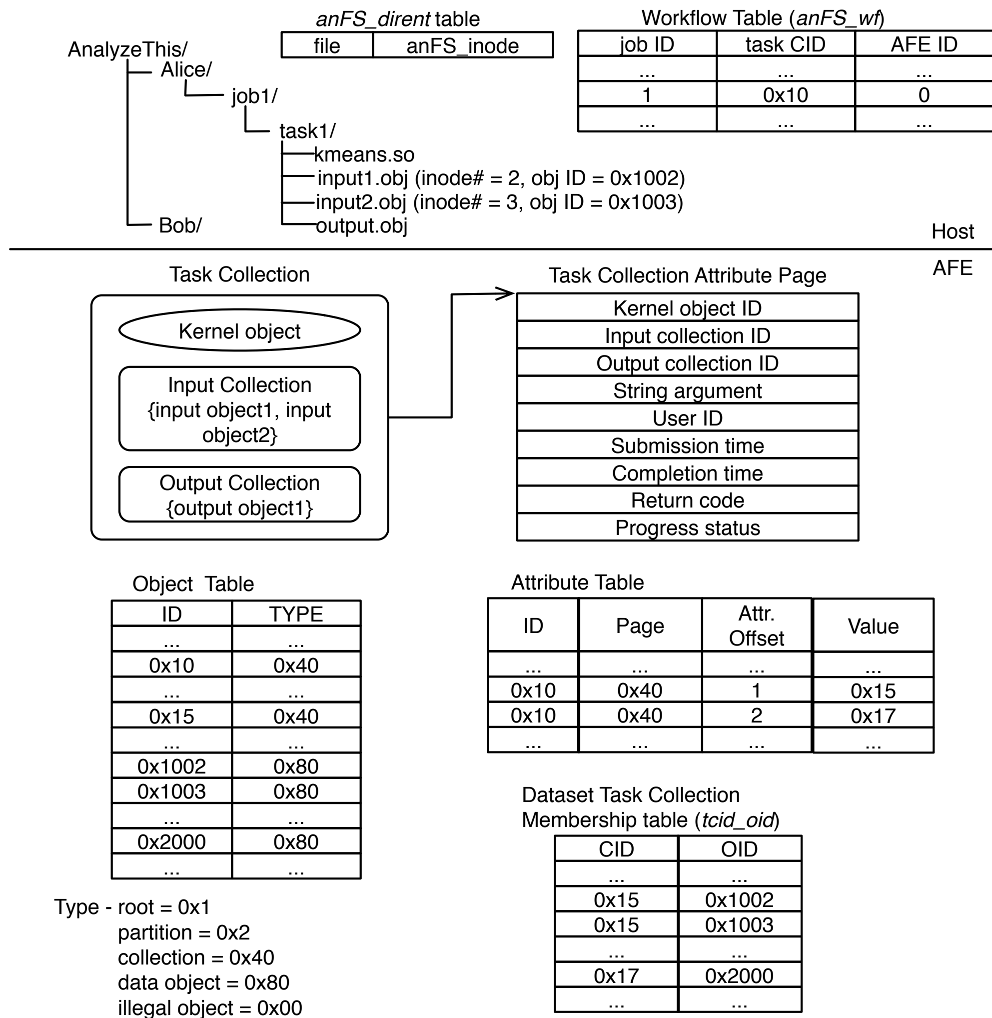
We realize the analysis object abstraction using the object storage device (OSD) protocol. The OSD protocol provides a foundation for us to build on, by supporting host to AFE communication and by enabling an object container-based view of the underlying storage. However, neither OSD nor OSD-2 supports analysis-awareness specifically. We use an open source implementation of the OSD T10 standard, Linux open-osd target [47], and extend it further with new features to implement the AFEs. We refer to our version of the OSD implementation as "AFE-OSD" (Figure 4.1). Our extensions are as follows: (i) *Mini Workflow* supports the execution of entire branches of an analysis workflow that is handed down by the higher-level Workflow Scheduler on the host; (ii) *Task Tracker* tracks the status of running tasks on the AFE; (iii) *AFE Status* checks the internal status of the AFEs (e.g., load on the controller, capacity, wear-out), and makes them available to the higher-level Workflow Scheduler to enable informed scheduling decisions; (iv) *Lineage Container* captures the lineage of the executed tasks; and (v) *Lightweight Database Infrastructure* supports the above components by cataloging the necessary information and their associations.

Mini Workflow Engine: The AFE-OSD initiator on the host submits the mini DAG to the AFE-OSD target. The mini DAG represents a self-contained branch of the workflow that can be processed on an AFE independently. Each mini DAG is composed of a set of tasks. A task is represented by an analysis kernel, and a series of inputs and outputs. The host dispatches the mini DAGs to the AFEs using a series of `ANALYZE_THIS` execution commands, with metadata on the tasks. Once the AFE receives the command, it will insert the task into a FIFO task queue that it maintains internally. The AFE will also create an *analysis task collection* for every task, an encapsulation to integrally tie together the analysis kernel, its inputs and outputs (*Task Collection* and the Task Collection Attribute Page are represented in the bottom half of Figure 4.2). The task collection is a new collection primitive we have implemented in the AFE-OSD. As we noted earlier, inputs and outputs can comprise of thousands of files. To capture this notion, we create a *linked collection* encapsulation for input and output datasets (using an existing *Linked* collection primitive), which denotes that a set of files are linked together and belong to a particular dataset.

Kernel Executor: The kernel executor is a multi-threaded entity that checks the task queue and dispatches the tasks. We have only used one core per AFE controller, but the design allows for the use of many cores.

Task Tracker: The Task Collection and the Task Collection Attribute page provide a way to track the execution status of a task and its execution history, i.e., run time. Each task collection has a unique task id. The host can check the status of a running task by reading an attribute page of its task collection using the `get_attribute` command and the task id. The workflow scheduler on the host also queries the AFE for the execution history of analysis kernels, to get an estimate of run times that are then used in scheduling algorithms, e.g., *Minimum Wait* (in Section 4.2).

AFE Status: A host can use an AFE's hardware status for better task scheduling. To this end, we have created a *status object* to expose the internal information to the host. The status object includes AFE device status such as wear-out for the flash, resource usage status for controller and



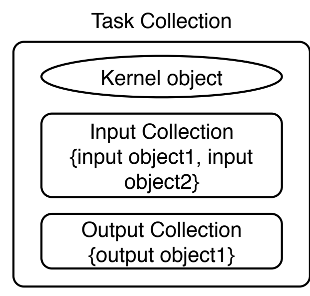


Figure 4.2: Analysis object abstraction implemented using database engine. CID: task collection id, OID: object id, Attr. offset: an offset in the attribute page.

memory, the AFE task queue details, and Garbage Collection status. The AFEs are configured to periodically update a local status object, which can then be retrieved by the host as needed. Thus, a host can check the status of the device by sending a `get_attribute` command on the status object using its object id.

Lineage Container: Lineage information of tasks and data objects are maintained in an AFE's internal database. The lineage container helps answer provenance queries (more details in Section 4.4).

Lightweight Database Infrastructure: We use a lightweight database infrastructure (Figure 4.2), using SQLite [48], to implement analysis-aware semantics into the storage system. One approach is to have the host maintain all of the analysis workflow, data, and analysis-semantics. However, such a centralized approach is not resilient in the face of host crashes. Instead, we implement a decentralized approach, wherein the host (the FUSE layer) and the AFEs (the AFE-OSD Target) maintain relevant information and linkages to collectively achieve the analysis abstraction.

The host database table (*anFS_wf*) maintains high-level information about the analysis workflow, e.g., mini DAGs (job ID), the associated tasks (task collection ID), and the AFE ID on which to run the task. For example, in Figure 4.2 user Alice runs a job (id = 1) and executes an analysis task, *kmeans* (CID = 0x10), on AFE (id = 0). The local AFE database tables store detailed metadata on all objects, namely mini DAGs, task collections, input and output datasets, their attributes and associations.

Each AFE manages three tables. The *Object table* is used to identify the type of an object, e.g., whether it is a task collection, or a data object. For each object, it maintains object identifiers, and object types. The *Dataset TaskCollection Membership table, tcid_oid,* manages the membership of data objects to task collections. Multiple data objects can belong to a task collection (e.g., multiple inputs to a task) or a data object can be a member of multiple task collections (e.g., a given dataset is input to multiple tasks). The *Attribute table* manages all the attributes of data objects and task collections (e.g., those represented in the Task Collection Attribute Page). Each attribute (or record) in the attribute table is defined using a data object or task collection id, page number, and attribute number inside the Attribute Page. Given this metadata, a host can query information on the tasks and their associated datasets. For example, given a task collection of 0x10 and an index into the attribute page, 1 (to refer to input datasets), the attribute table points to a value of 0x15, which can be reconciled with the *tcid_oid* table to obtain the input datasets 0x1002 and 0x1003.

4.2 Workflow Engine

We have built a *workflow engine* within AnalyzeThis, to orchestrate both the placement of data objects as well as the scheduling of analysis tasks across the AFEs. The scheduler is implemented in the FUSE file system (anFS). Once a user submits the job script via anFS, it distinguishes this request from a normal I/O request. The script is delivered to the scheduler which parses it to build a directed acyclic graph (DAG), schedules the tasks, and sends the execution requests to the AFEs via the OSD protocol. The vertices in the DAG represent the analysis kernels, and inputs and outputs represent incoming and outgoing edges. The scheduler decides which branches (mini DAGs) will run on which AFEs based on several heuristics. While the mapping of a mini DAG to AFE is determined apriori by the scheduler, the tasks are not dispatched until the simulation outputs (analysis inputs) are written to AnalyzeThis. This is akin to the *smart folder* concept discussed in Section **??**. The analysis sequence is first registered with AnalyzeThis, and once the input data is

available the tasks are executed.

4.2.1 Workflow Description and DAG

A job script can be written in any language, however, in our implementation, we choose *Libconfig* [49], a widely used library for processing structured configuration files. Listing 4.1 shows an example of a job that finds the maximum value in each input file. Each tasklet is represented by the input and output object lists, and a kernel object that operates on the input objects. Any data dependencies among the tasklets are detected via a two-pass process. The scheduler parses a job script in two passes. In the first pass, the scheduler examines each task in the script and inserts the task record (key: output file, value: task) into a hash table. In the second pass, the scheduler examines the input files of each task in the job script. When an input file is found in the hash table, the examined task is dependent on the output of the task in the hash table. If the input file is not found in the hash table, the scheduler checks if the input file already exists in the file system. If the file does not exist, the job script is considered to be invalid. In this way, the dependencies among the tasks can be resolved. The dependency information is used to build a DAG. In the following example, `getmax.reduce` cannot be launched until `getmax.1` and `getmax.2` produce their output objects. Therefore, the overall performance of AnalyzeThis depends on the efficient scheduling of the analysis kernels on the AFE array.

Listing 4.1: An example job script

```
name = "getmax";
workdir = "/scratch/getmax/";
tasks = (
  { name = "getmax.1"; kernel = "getmax.so";
    input = [ "1.dat" ]; output = [ "1.max" ]; },
  { name = "getmax.2"; kernel = "getmax.so";
    input = [ "2.dat" ]; output = [ "2.max" ]; },
  { name = "getmax.reduce"; kernel = "mean.so";
    input = [ "1.max", "2.max" ];
    output = [ "max.dat" ]; }
);
```

4.2.2 Scheduling Heuristics

The design of the workflow scheduler is driven by two objectives: (1) minimizing the overall execution time, and (2) reducing the data movement across the AFEs. We point out that minimizing data movement is critical as uncontrolled data movement may cause early wear-out of the SSDs and increase in the energy consumption [50]. An opportunity here is that the overall execution time of the analysis workflow can be minimized by exploiting the available parallelism therein. With this in mind, we device several strategies.

Round-robin: A simple round-robin approach schedules tasks as soon as their dependency requirements are met and ensures a homogeneous load-distribution across all AFEs, in a best-effort manner since the tasks are scheduled without apriori information about their execution time. It picks the next available AFE in a round-robin fashion to balance the computational load. Unfortunately, the technique suffers from excessive data movement because it does not account for the amount of data to be moved. Consider a simple example, where the input data for the next scheduled task resides on a particular AFE, but the AFE controller is busy executing another task. In this case, the round-robin strategy will schedule the task on another idle AFE controller, causing data-movement, potentially in favor of a shorter execution time and load balance across the AFE controllers.

Input Locality: To minimize the data movement across the AFEs, this heuristic schedules tasks based on input locality. Tasks are scheduled on an AFE where maximum amount of input data is present. The scheduler maintains this information in memory during a job run, including the size and location of all involved files. For example, consider a task that requires two inputs, say a and b (where $\texttt{size_of}(\texttt{a}) > \texttt{size_of}(\texttt{b})$) that reside on $AFEa$ and $AFEb$, respectively. The scheduler will execute the analysis task on AFE_a, which stores the larger input, even if the controller on AFE_a is executing another task and the controller on AFE_b is idle. Input-locality favors a reduction in data movement to performance (execution time). In our experiments with real, complex scientific workflows, we observed that this scheduling policy is effective in reducing the data-movement. However, it does increase the execution time.

Minimum Wait: To reconcile execution time and data movement, we propose to explicitly account for data transfer time and queuing delays on the AFE controllers. The heuristic takes two inputs: list of all available AFEs and the tasks to be scheduled next. The scheduler maintains information about the jobs currently queued on each AFE, their expected finish time and the size of the input file(s) for the task to be scheduled next. The scheduler iterates over each AFE to estimate the minimum wait time for the task to be scheduled. For each AFE, it calculates the queue wait time (due to other jobs) and data transfer time to that particular AFE. It chooses the AFE for which the sum of these two components is minimum. The minwait scheduler maintains and updates the "expected free time" of each AFE using the runtime history of jobs. When a task is ready to be executed, the scheduler calculates the expected wait time of the task for every AFE. The expected wait time at an AFE is calculated as: "expected free time" at the AFE, plus the expected data transfer time (estimated using the input file size and AFE location). The scheduler assigns the task to an AFE that is "expected" to have the minimum wait time.

4.3 anFS File System Interface

The functionalities of AnalyzeThis are exposed to the hosts via a specialized file system interface, "anFS." Since the workflows operate on but do not modify the HPC simulation data, anFS is designed as a write-once-read-many file system. It provides standard APIs such as `open()`, `read()`,

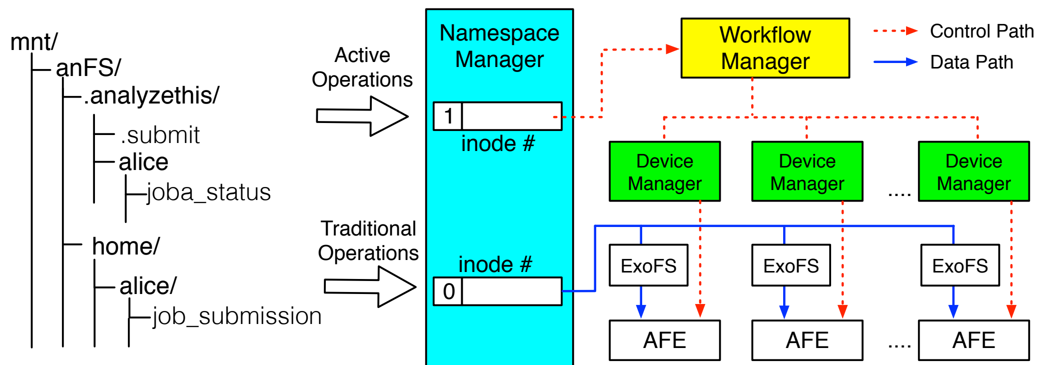


Figure 4.3: anFS architecture shows the namespace, workflow, device manager, and exoFS components with the request flow.

and `write()`, as well as support special virtual files (SVFs), serving as an interaction point, e.g., to submit and track jobs, between users and AnalyzeThis.

Figure 4.3 shows the overall architecture of anFS. It is implemented using the FUSE user-space file system, and can be mounted on the standard file system, e.g., `/mnt/anFS/`. anFS is composed of the following components. The *Namespace Manager* consolidates the array of available AFEs, and provides a uniform namespace across all the elements. The *Workflow Manager* implements the workflow engine of AnalyzeThis (Section 4.2). The *Device Manager* provides the *control path* to the AFEs, implementing AFE-OSD (Section 4.1.1) to allow interactions with the AFEs. Finally, the *ExoFS (Extended Object File System) layer* provides the *data path* to the AFEs, and is based on the ext2 file system.

Namespace: anFS exposes a consolidated standard hierarchical UNIX namespace view of the files and SVFs stored on the AFEs. To this end, we include in our host-side metadata table (Section 4.1.1) additional information associated with every stored object as shown in Figure 4.2, as well as information to track the AFEs on which the objects are stored. For example, there is an AFE identifier and an object identifier associated with every inode of a file stored by anFS. All file system operations are first sent to the Namespace Manager that consults the metadata to route the operation to an appropriate AFE. To manage the large amount of metadata that increases with increasing number of files, provide easy and fast access, and support persistence across failures etc., we employ the SQLite RDBMS [48] to store the metadata. We note that anFS implements features such as directories, special files, and symbolic links, entirely in the metadata database; the AFEs merely store and operate on the stored data objects. In contrast to striping files across storage elements in PFS for fast parallel access to an external processing element, anFS stores an entire file on a single AFE to facilitate on-element analysis and reduce data movement. The placement of a file on an AFE is either specified by the workflow manager, or a default AFE (i.e., inode modular number-of-AFEs) is used.

Data and Control Path: To provide a data path to the AFEs, anFS uses exoFS, an ext2-based

file system for object stores. anFS stores regular data files via the exoFS mount points, which are created one for each AFE. For reads and writes to a file, anFS first uses the most significant bit of the 64-bit inode number to distinguish between a regular file (MSB is 0) and a SVF (MSB is 1). For regular files, the Namespace Manager locates the associated AFE and uses exoFS to route the operation to the AFEs as shown in Figure 4.3. Upon completion, the return value is returned to the user similarly as in the standard file system. To provide a control path for active operations, anFS intercepts the files and routes it to the Workflow Manager, which uses the Device Manager to route the operations to the AFEs using the OSD library for further anlaysis and actions. In the OSD standard, collections and data objects share the same id space. Since collections are created by anFS for active operations, and data objects are created by exoFS, we need to avoid id collisions. To address this issue, we have modified the exoFS to export a `/sys` entry (sysfs in Linux). Each time anFS needs to assign a collection id, it obtains the id from the sysfs entry and id assignments in both filesystems (anFS and exoFS) are reconciled.

Active File Operations — Job Submission: anFS supports SVFs to allow interaction between users and AnalyzeThis operations, e.g., running an analysis job, checking the status of the job, etc. Specifically, we create a special mount point (`.analyzethis`) under the root directory for anFS (e.g., `/mnt/anFS/`), which offers similar functionality as that of `/proc` but for workflow submission and management (Figure 4.3). To submit a job, e.g., JobA, the user first creates a submission script (`/home/alice/joba-submission`) that contains information about how the job should be executed and the data that it requires and produces. Next, the job is submitted by writing the full path of the submission script to the submission SVF, e.g., by using `echo /home/alice/joba-submission > /mnt/anFS/.analyzethis/alice/submit`. This SVF write is handed to the Workflow Manager for processing, which parses the script, assigns a unique opaque 64-bit job handle to the script, and takes appropriate actions such as creating a task schedule, and using the appropriate Device Manager thread to send the tasks to the AFEs. The Workflow Manager also updates a per-user active job list, e.g., SVF `/mnt/anFS/.analyzethis/alice/joblist` for user alice, to include the job handle for the newly submitted job. Each line in the joblist file contains the full path of the submission script and the job handle. Moreover, the Workflow Manager also monitors the task progress. This information can be retrieved by the user by reading from the job handle SVF `/mnt/anFS/.analyzethis/alice/joba-status`. When the user accesses the job handle, the request is directed to the Device Manager thread for the AFE, via the Workflow Manager. The Device Manager thread sends the `get_attribute` command via the AFE-OSD protocol to the *Task Tracker* in the Mini Workflow Engine on the AFE to retrieve the status of the jobs.

Supporting Internal Data Movement: Ideally, AnalyzeThis will schedule a task to an AFE that also stores the (*most*) data needed by the task. While we attempt to minimize data movement through smart heuristics, there is still the need to move data between AFEs as a perfect assignment is not feasible. To this end, anFS may need to replicate (or move) data from one AFE to another by involving the host. However, this incurs added overhead on the host. In the future, direct PCI to PCI communication can help expedite these transfers.

Data Availability/Reliability AnalyzeThis is designed to be an in-situ data analysis system, where

the needed data is buffered in the active flash elements and is always backed by an underlying distributed file system such as Spider [6]. Thus, we do not employ additional data redundancy in the anFS and rely on the underlying file system to recover data for any failed AFE. An optimization in this context is that if the data has been replicated for better scheduling as discussed above, we can simply start using the surviving replica instead of recovering data from the underlying layer.

Data and Workflow Security anFS ensures data security and privacy across multiple users by leveraging the underlying kernel-level and OSD standards [25]. For protecting the stored data, anFS relies on the T10 OSD2 standard, where an AFE maintains the ownership of objects as object attributes. When a data item is stored on the AFE, the protocol also provides the kernel-level user-id of the data owner, which is then stored in the AFE-level object ownership metadata automatically by the device. When the data is accessed, the user-id information is provided along with the request, and the OSD2 protocol ensures that only the allowed user(s) are given access to a stored data item. Similarly, when a task is scheduled on an active flash element, it is associated with the user-id information. The task must present this information to the OSD every time it needs to access a data object, and only accesses from tasks associated with allowed users are permitted. Similarly, SVFs are protected using user-specific sub-directories, e.g., Alices jobs are under `.anFS/alice/`, and the associated POSIX ACLs are set such as only the owner can access the file. This allows anFS to support multi-user workflow submission and management in a secure manner.

File sizes and SSD capacity We store an entire file on an OSD, which can be problematic if a single file is larger than the size of an available OSD. We argue that the capacity of modern SSDs is quite large, and will be able to handle most of the files without a problem. In the unlikely event that a larger file than a single SSDs capacity needs to be stored, we propose to partition (or re-transform if a specialized file format is employed) a file into smaller files that can be handled as discussed above. Moreover, we also have to update the submission script to ensure that the multiple partitions can be referenced correctly. While it is possible to do such updates automatically, in our current system, we rely on the users to update their scripts should a file partitioning becomes necessary.

4.4 Provenance

AnalyzeThis tracks the lineage of the data produced as a result of a workflow execution at very minimal cost. This allows the user to utilize the intermediate data for future analysis. We have implemented provenance support on top of the distributed database infrastructure (Figure 4.2) between the host (workflow table, *anFS_wf*) and AFEs (Dataset task collection membership table, *tcid_oid*). Recall that *anFS_wf* stores information about the task and the AFE on which the task is executed; *tcid_oid* stores the task collection to data object mapping and will also need to be maintained on the host. Upon receiving a provenance query regarding a dataset, AnalyzeThis searches the *anFS_dirent* table to get the *anFS_inode* of the file, which is used to get the object id. The object id is then used to retrieve the task collection id from the *tcid_oid* table. The task

collection id is used to obtain the AFE id from the *anFS_wf* table. Alternatively, if *tcid_oid* is not maintained on the host as well, we can broadcast to the AFEs to determine the task collection id for a data object id. Further analysis of the lineage is performed on that AFE. Using the task collection id and the attribute page we get the task collection attribute page number. Using the predefined attribute offset all the information regarding the task is fetched. The task provenance from multiple AFEs is merged with similar job-level information that is maintained at the host in *anFS_wf* table.

While not part of the implementation, it can be beneficial to write the provenance information (`.prov`) of the intermediate and final dataset when they are flushed to the PFS so the lineage is part of the dataset, even beyond its lifetime in AnalyzeThis. This is useful in supporting newer data analysis on the intermediate data in the future, without having to rerun all of the workflow. For example, before the execution of a DAG, AnalyThis can query the `.prov` files in the PFS to see if a particular mini DAG or a branch has been run before, and reuse the result data.

Chapter 5

Evaluation

5.1 Experimental Setup

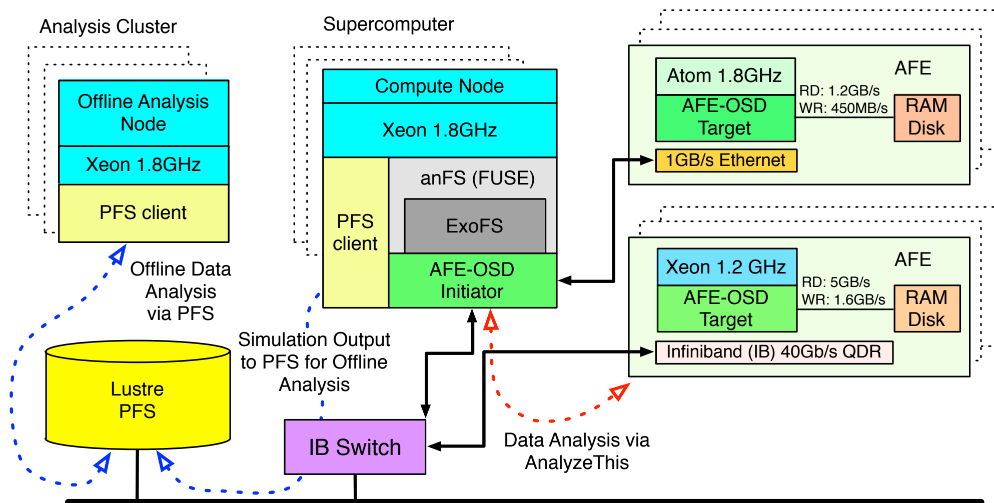
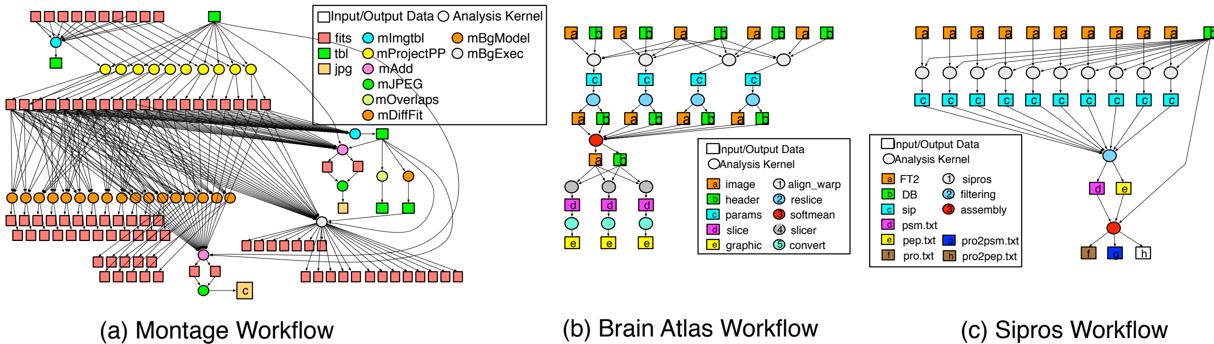


Figure 5.1: AnalyzeThis testbed.

5.1.1 Hardware Platform

Our emulation testbed (Figure 5.1) comprises of a compute node, where the application simulation runs, and to which the AFEs are connected; the networked servers that emulate the AFEs; and the analysis node, which runs the offline data analysis. For the compute and analysis node, we used a 1.8 GHz Intel Xeon processor. We emulated the AFEs using Atom or Xeon servers with RAM disks, to mimic the flash controller and the internal flash chips with high I/O bandwidth to the



(a) Montage Workflow (b) Brain Atlas Workflow (c) Sipros Workflow

Figure 5.2: The DAGs representing the workflows.

controller. The Atom-based AFEs use a single 1.7 GHz Atom processor as the controller, a 3 GB RAM disk as the flash chip, and a 1 Gbps Ethernet connection to the simulation compute node. The Xeon-based AFEs use a clocked-down 1.2 GHz Xeon processor, a 64 GB RAM disk, and a 40 Gbps IB QDR connection to the compute node. The Xeon-based AFEs enable the emulation of more powerful flash controllers. Also, since in reality, the AFEs will be connected locally to the compute node via a high I/O bandwidth PCIe interface, the IB network emulates this connection by providing a bandwidth of 5 GB/s. (Note that a PCIe Gen 3.0 x8 offers 8 GB/s.) All servers run the Linux kernel 2.6.32-279. The offline data analysis is conducted via a Lustre 2.4 PFS with one OSS/OST and a SAS 10K RPM 1 TB drive.

5.1.2 Software

AnalyzeThis has been implemented using 10 K lines of C code. We used an open source implementation of the OSD standard, Linux open-osd [47], and extended it for our purposes. Our AFE implementation is an extension to the OSD emulator component of open-osd, which is based on the Linux iSCSI Target [51]. The active task processing in the AFE is emulated by allocating a dedicated thread that runs the tasks sequentially. We have also made modifications to the open-osd driver to support the host-side requirements. anFS has been implemented using the FUSE [52] framework. The user space implementation allows us to easily integrate the workflow manager in the system. anFS keeps track of its metadata using the SQLite RDBMS [48].

5.1.3 Scientific Workflows

We used three real-world, complex scientific workflows – Montage [53], Brain Atlas [54] and Sipros [55] (Figure 5.2 and Table 5.1).

The Montage workflow [53] creates a mosaic with 10 astronomy images. It uses 8 analysis ker-

	Input	Intermediate (MB)	Output	Total	Object (#)
Montage	51	222	153	426	113
Brain Atlas	70	155	20	245	35
Sipros	84	87	1	172	45

Table 5.1: Workflow input, output and intermediate data size.

nels, and is composed of 36 tasks, several of which can be parallelized to run on the AFEs. The Brain Atlas workflow [54] creates population-based brain atlases from the fMRI Data Center's archive of high resolution anatomical data, and is part of the first provenance challenge used in our provenance evaluation. The Sipros workflow runs DNA search algorithms with database files to identify and quantify proteins and their variants from various community proteomics studies. The workflow consists of 12 analysis tasks, and uses three analysis kernels.

5.2 AnalyzeThis Performance

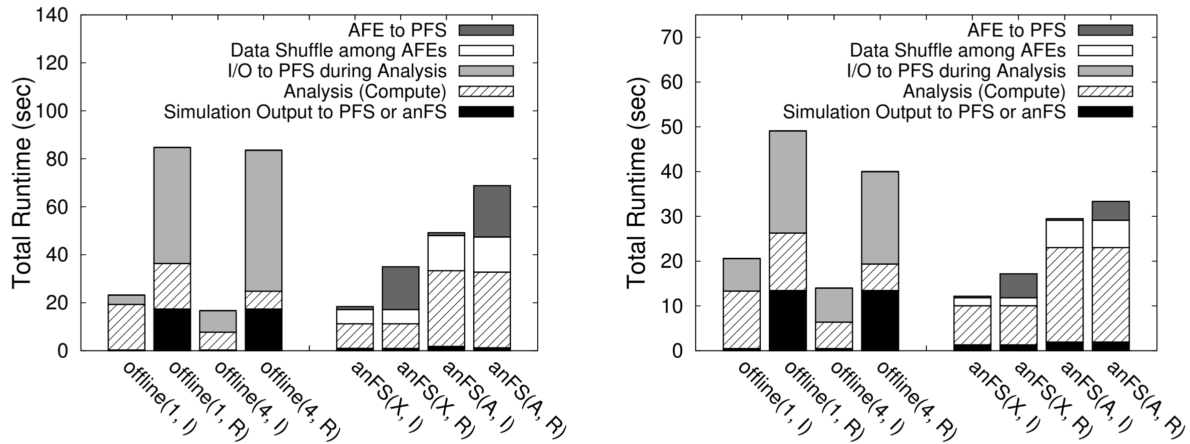
5.2.1 anFS Performance

Table 5.2 shows the sequential performance for reading and writing a 1 GB dataset from a single compute node to anFS with one AFE, and the Lustre PFS with one OSS/OST and a stripe size of 1 MB.

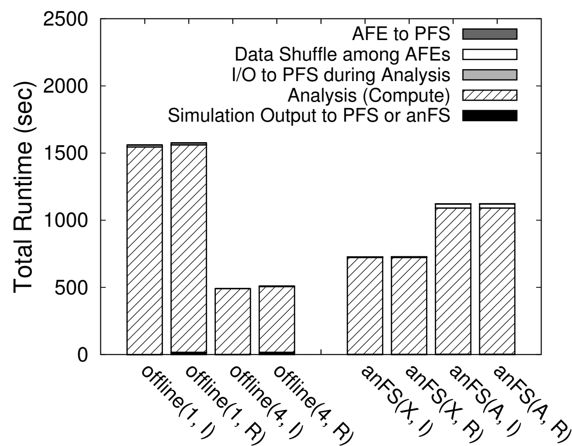
	PFS_{Ideal}	anFS(IB)	anFS(1Gb)
Read (MB/s)	84.6	353.8	119.0
Write (MB/s)	79.9	118.0	79.0

Table 5.2: Avg. read/write rate for 1 GB dataset.

The table shows the ideal PFS rate when there is no other workload present, and anFS rates using both IB and Ethernet. Both *anFS(IB)* and *anFS(1Gb)* perform better than *PFS_ideal* for reads and writes, with *anFS(IB)* providing 3x read throughput compared to the other two. Note that the anFS rates include the FUSE, exoFS and OSD overhead as well. *PFS_ideal* delivered around 80 MB/s for reads and writes on a quiet Lustre PFS. To draw an analogy with a real-world HPC setting, note that the Titan supercomputer's shared PFS (one of the fastest in the world) offers a peak throughput of 1 TB/s to its 18,688 compute nodes (56 MB/s/node). However, in reality, applications only realize much less than an order of magnitude (10 GB/s - 100 GB/s) I/O throughput due to contention [56]. To mimic this real-world HPC scenario, we show end-to-end workflow results based on both the ideal and an order of magnitude reduced (8 MB/s) PFS measurement. Since anFS is local to the supercomputer's compute node, where timesharing is not allowed, there is no contention therein.



(a) Montage Workflow (b) Brain Atlas Workflow



(c) Sipros Workflow

Figure 5.3: End-to-end workflow performance of offline (one and four nodes) and AnalyzeThis (Xeon and Atom AFEs). AnalyzeThis used four AFEs and the round-robin scheduling. Both ideal and real PFS bandwidth based results are shown.

5.2.2 End-to-End Workflow Experiments

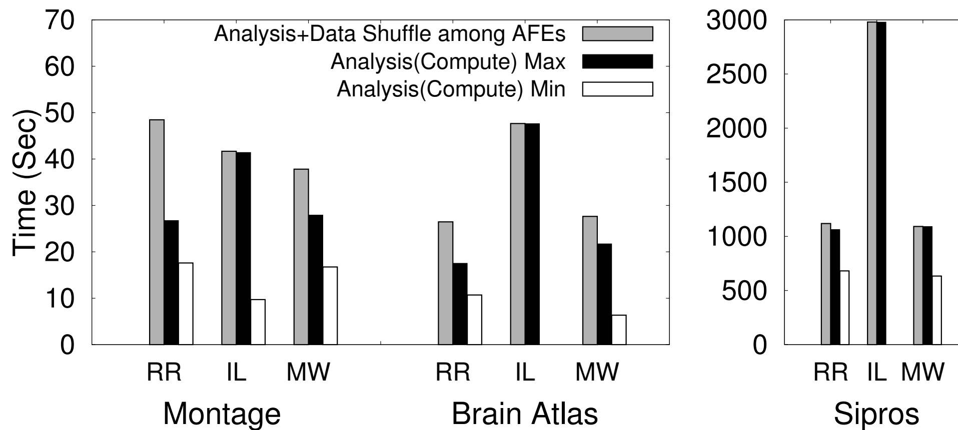
We conducted the Montage, Brain, and Sipros end-to-end workflow tests via both *Offline* and *AnalyzeThis* approaches (Figure 5.3). A breakdown of the end-to-end workflow runtime comprises of the following parts: (i) time to write the simulation output to either PFS or anFS, (ii) computation time of the analysis workflow, (iii) I/O time to write the intermediate output to PFS during analysis

(Offline case), (iv) data shuffling time among the AFEs, and (v) time to write the final analysis output from the AFE to the PFS. We compared the following scenarios. (i) Offline analysis using a single node with ideal (I) and real-world (R) PFS bandwidth, $offline_{(1,I)}$, $offline_{(1,R)}$, (ii) Offline analysis using four nodes with ideal and real-world PFS bandwidth, $offline_{(4,I)}$, $offline_{(4,R)}$, (iii) AnalyzeThis using four Xeon-based AFEs with IB, under ideal and real-world PFS bandwidth, $anFS_{(X,I)}$, $anFS_{(X,R)}$, and (iv) AnalyzeThis using four Atom-based AFEs with Ethernet, under ideal and real-world PFS bandwidth, $anFS_{(A,I)}$, $anFS_{(A,R)}$. The AnalyzeThis experiments used the round-robin scheduling algorithm.

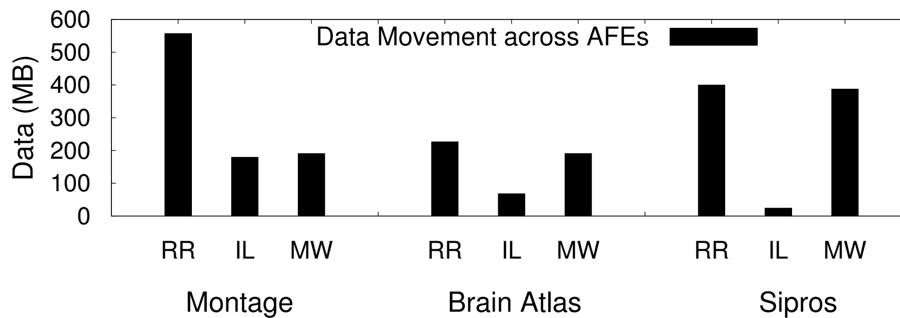
First, our tests used an ideal PFS bandwidth. We notice that $offline_{(1,\, I)}$ is worse than *AnalyzeThis* with Xeon-based AFEs, $anFS_{(X,\, I)}$, for all workflows due to the lack of parallelism for task execution (Figure 5.2), even though the Xeon-based AFEs run at a lower clock than the offline analysis node. Recall that we used four AFEs for *AnalyzeThis*. Compared to *AnalyzeThis* with Atom-based AFEs, $anFS_{(A,\, I)}$, $offline_{(1,\, I)}$ shows much lower runtimes for Montage and Brain workflows, but a higher runtime for the Sipros workflow. This is because the Montage and Brain workflows are "compute-bound", where the CPU speed is critical, whereas Sipros is "memory-bound", which requires lower memory latency. Both Xeon and Atom-based AFEs have different memory latencies, and the Xeon-based AFE is a faster controller than the Atom-based AFE.

Next, we used a realistic PFS bandwidth scenario, with an order of magnitude reduction in I/O throughput (8 MB/s). In the Montage and Brain workflow experiments for *offline* ($offline_{(1,\, I)}$ vs. $offline_{(1,\, R)}$), the time to write the simulation output and the analysis intermediate output to the PFS consume the bulk of the total execution time, whereas for *AnalyzeThis* ($anFS_{(X/A,\, I)}$ vs. $anFS_{(X/A,\, R)}$), there is only an increase in the time to write the final analysis output from the AFE to the PFS. Due to an increased I/O time to the PFS, $offline_{(1,\, R)}$ has higher runtimes than *AnalyzeThis* with Atom-based AFEs. On the contrary, Sipros results show little difference in performance between ideal PFS and realistic PFS conditions. This is because the Sipros workflow is dominated by the computational time for analysis. The results indicate that *offline*'s performance is heavily affected by the PFS performance, whereas *AnalyzeThis* is relatively unperturbed by the PFS rate. Also, the time for data shuffle and I/O to PFS will only improve for the Atom-based AFEs on better connectivity than the Gbps Ethernet.

We examined the impact of multiple nodes for *offline* compared to *AnalyzeThis*. We used four nodes for *offline* since we used four AFEs for *AnlyzeThis*. As expected, the compute time reduced due to more nodes ($offline_{(1,I/R)}$ vs. $offline_{(4,I/R)}$). However, surprisingly, the time taken to do I/O to the PFS during analysis slightly increased. For instance, the Montage result for $offline_{(4,R)}$ compared to $offline_{(1,R)}$ shows an increase of 21% for the time taken to perform I/O to the PFS. This increase can be attributed to the intra-job I/O contention. Finally, even as we add parallelism to *offline* analysis, $anFS_{(A,R)}$ outperforms $offline_{(4,R)}$ by up to 17% for Montage and Brain workflows.



(a) Total run-time and AFE core utilization



(b) Data movement

Figure 5.4: Performance of scheduling heuristics with Atom-based AFEs.

5.2.3 Impact of Scheduling Heuristics

In Figure 5.3, we used the round-robin (RR) scheduling technique for *AnalyzeThis*. In Figure 5.4, we evaluate the performance of RR, input locality (IL), and minimum wait (MW) algorithms using Atom-based AFEs, based on AFE utilization and data movement. Figure 5.4(a) compares the sum (first bar) of the computation time of the analysis workflow and the data shuffling time among the AFEs against the AFE utilization time (other two bars). AFE utilization is denoted by the slowest (second bar) and the fastest (third bar) AFEs, and the disparity between them indicates a load imbalance across the AFEs. The smaller the difference, the better the utilization. Figure 5.4(b) shows the amount of data shuffled between the AFEs. An optimal scheduling technique should strike a balance between runtime, data movement, and AFE utilization.

Overall RR shows a balanced load distribution across the AFEs with the least variability in utilization, however, as expected, it incurs the most data movement. IL can improve runtime by significantly reducing data movement, however it may degrade the overall performance due to

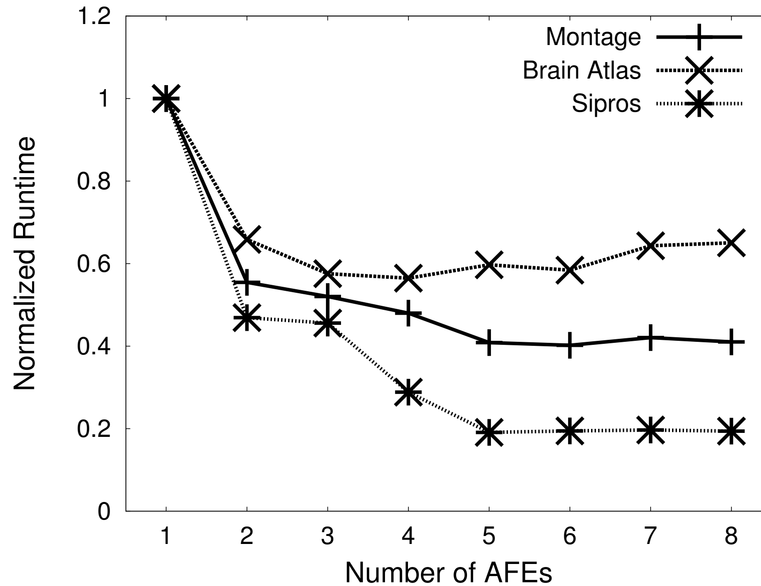


Figure 5.5: AFE scaling experiments for AnalyzeThis.

inefficient load distribution. For Montage, IL achieves a lower runtime than RR due to a significant decrease in data movement, but it shows significantly higher runtimes than RR for Brain and Sipros. In fact for IL, we observed in Montage that the slowest AFE was assigned 21 tasks among 36 tasks; in Brain, only two AFEs out of four executed the all of the tasks; and in Sipros, only one AFE was used during analysis. MW performs best in reconciling AFE utilization and data movement cost. For Montage, MW shows a 10% lower runtime than IL by incurring a 6% increase in data movement. For Brain, RR and MW show very close runtimes, but MW further reduces the data movement cost of RR by 15%, with less core utilization, suggesting that it is likely to be more energy-efficient. For Sipros, MW shows a 2.4% lower runtime than RR while reducing the data movement cost by 3%.

5.2.4 Scaling Experiments

We performed scalability experiments for *AnalzeThis* by increasing the number of Atom-based AFEs.

Figure 5.5 shows the results with MW. Interestingly, we observe that the overall performance scales only up to a certain number of AFEs, since the maximum task parallelism in the workflow can limit the performance gain. In the Montage workflow, for instance, `mProjectPP` is the most time-consuming kernel, used by ten analysis tasks. After five AFEs, at least one AFE will run one more `mProjectPP` than others. Thus, there is little improvement in performance after five AFEs. Likewise, Brain and Sipros scale only up to four or five AFEs, respectively. Also, each workflow

shows a different speed-up curve with the increase in AFEs, depending on its task dependency, number of kernels, inputs and outputs. For instance, Sipros shows a perfect linear speedup up to five AFEs. This is because the less complex Sipros DAG (Figure 5.2(c)) allowed for more parallelism in task execution with the increase in AFEs than others. In some cases, however, the increased data movement cost with more AFEs can degrade the performance. For instance, Brain shows a slightly higher runtime in going from four to eight AFEs, due to the increased data movement.

5.2.5 Provenance Performance

We used the Brain Atlas workflow (Figure 5.2(b)) and the provenance queries from the first provenance challenge [54] to evaluate AnalyzeThis. We ran the workflow on AnalyzeThis and stored the lineage metadata. The provenance test consists of five queries, where Query 1 finds the provenance data up to the start of a job for a file; Query 2 finds the provenance data up to a task name for a file; Query 3 finds the provenance details for the levels within a DAG for a file; Query 4 finds all the invocations of a task that ran with the certain parameters and on a specific date; and Query 6 finds all output images produced by a task when another task was executed with certain arguments.

Query	**Centralized** (sec)	**Decentralized** (sec)
1	12.930	8.180
2	1.750	2.830
3	13.746	10.720
4	0.410	3.400
6	7.410	7.640

Table 5.3: Response time for five provenance queries.

In addition to the decentralized approach (Section 4.4), we implemented a centralized technique, where all of the lineage information is maintained on the host. For both approaches, one million entries were added to and retrieved from all the host side tables. For the decentralized approach, four AFEs were used and each had 0.25 million entries in all the database tables. For three out of five cases (Table 5.3), centralized performed better in terms of the query response time. Even though the query execution is distributed to multiple AFEs, there is no improvement in the processing time for some queries as the host-side provenance information still needs to be parsed. Reduction in query time for the decentralized case is primarily due to the parallelization opportunity available in the workflow. On the other hand, the centralized approach offers no fault tolerance in the event of a host crash. Thus, decentralization may be desirable even if the query time is higher in certain cases.

Chapter 6

Conclusions and Future Works

The need to bridge the gap between workflow and storage systems is urgent. The chaining of simulations and data analyses via a PFS will not work at future data production rates. With AnalyzeThis, we have shown how analysis-awareness can be built into each and every layer of a storage system. The concepts of building an analysis object abstraction atop an active flash fabric, integrating a workflow scheduler with the storage, and exposing them via a /proc-like file system bring a fresh perspective to purpose-built storage systems. Our evaluation of AnalyzeThis shows that the approach is viable, and can be used to capture complex scientific workflows efficiently. Furthermore, while we have focused on the scientific workflow, AnalyzeThis is equally applicable to supporting other analysis workflows such as those encountered in the enterprise and big data applications.

Bibliography

[1] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[2] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, 2003.

[3] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters.," in *FAST*, 2002.

[4] "Introducing Titan." `https://www.olcf.ornl.gov/titan/`.

[5] "Top 500 Supercomputer sites." `http://www.top500.org/`.

[6] G. Shipman, D. Dillow, S. Oral, and F. Wang, "The spider center wide file system: From concept to reality," in *Proceedings, Cray User Group (CUG) Conference, Atlanta, GA*, 2009.

[7] R. L. Henderson, "Job scheduling under the portable batch system," in *Job scheduling strategies for parallel processing*, pp. 279–294, Springer, 1995.

[8] A. Guide, "Moab workload manager®," 2011.

[9] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, 2004.

[10] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[11] "DAGMan: A Directed Acyclic Graph Manager." `http://research.cs.wisc.edu/htcondor/dagman/dagman.html`.

[12] J. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. Hawkes, S. Klasky, W. Liao, K. Ma, J. Mellor-Crummey, N. Podhorszki, *et al.*, "Terascale direct numerical simulations of turbulent combustion using s3d," *Computational Science & Discovery*, vol. 2, no. 1, p. 015001, 2009.

[13] "TSUBAME Computing Services." `http://tsubame.gsic.titech.ac.jp/en`.

[14] S. M. Strande, P. Cicotti, R. S. Sinkovits, W. S. Young, R. Wagner, M. Tatineni, E. Hocks, A. Snavely, and M. Norman, "Gordon: design, performance, and experiences deploying and supporting a data intensive supercomputer," in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, p. 3, ACM, 2012.

[15] "Catalyst." `https://www.llnl.gov/news/newsreleases/2013/Nov/NR-13-11-01.html#.U2KJvlRdUal`.

[16] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, "stdchk: A checkpoint storage system for desktop grid computing," in *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, 2008.

[17] G. Bronevetsky and A. Moody, "Scalable i/o systems via node-local storage: Approaching 1 tb/sec file i/o," 2009.

[18] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: a checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[19] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman, "Active flash: Out-of-core data analytics on flash storage," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, 2012.

[20] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: a case for intelligent ssds," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013.

[21] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, 2013.

[22] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large scale data mining and multimedia applications," in *Proceedings of 24th Conference on Very Large Databases*, 1998.

[23] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)," *ACM SIGMOD Record*, vol. 27, no. 3, pp. 42–52, 1998.

[24] "Samsung SSD." `http://www.samsung.com/uk/consumer/memory-cards-hdd-odd/ssd/830`.

[25] R. O. Weber, "Information technologyscsi object-based storage device commands (osd)," *Technical Council Proposal Document*, vol. 10, pp. 201–225, 2004.

[26] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, D. D. E. Long, Y. Kang, Z. Niu, and Z. Tan, "Design and evaluation of oasis: An active storage framework based on t10 OSD standard," in *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, 2011.

[27] M. T. Runde, W. G. Stevens, P. A. Wortman, and J. A. Chandy, "An active storage framework for object storage devices," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, 2012.

[28] L. Qin and D. Feng, "Active storage framework for object-based storage device," in *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, 2006.

[29] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, 2006.

[30] C. Sar and P. Cao, "Lineage file system," *http://crypto. stanford. edu/cao/lineage. html*, 2005.

[31] "Virtual Data Toolkit." `http://vdt.cs.wisc.edu/`.

[32] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Explicit control a batch-aware distributed file system," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, 2004.

[33] S. Al-Kiswany, E. Vairavanathan, L. B. Costa, H. Yang, and M. Ripeanu, "The case for cross-layer optimizations in storage: A workflow-optimized storage system," *arXiv preprint arXiv:1301.6195*, 2013.

[34] A. Acharya, M. Uysal, and J. Saltz, "Active disks: programming model, algorithms and evaluation," *SIGPLAN Not.*, vol. 33, no. 11, p. 81–91, 1998.

[35] M. Singh and B. Leonhardi, "Introduction to the ibm netezza warehouse appliance," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011.

[36] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.-K. Liao, and A. Choudhary, "Enabling active storage on parallel i/o software stacks," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010.

[37] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[38] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008.

[39] "DOE exascale initiative technical roadmap." http://extremecomputing.labworks.org/hardware/collaboration/EI-RoadMapV21-SanDiego.pdf, 2009.

[40] "NetCDF Documentation." http://www.unidata.ucar.edu/packages/netcdf/docs.html.

[41] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *Proceedings of SC2003: High Performance Networking and Computing*, 2003.

[42] "HDF 4.1r3 User's Guide." http://hdf.ncsa.uiuc.edu/UG41r3_html/.

[43] "HDF5 - A New Generation of HDF." http://hdf.ncsa.uiuc.edu/HDF5/doc/.

[44] "HDF5 Tutorial: Parallel HDF5 Topics." http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/parallel.html.

[45] G. Velamparampil, "Data management techniques to handle large data arrays in HDF," Master's thesis, Department of Computer Science, University of Illinois, Jan. 1997.

[46] "Nexus." http://trac.nexusformat.org/code/wiki.

[47] "Open-OSD project." http://www.open-osd.org, 2013.

[48] "SQLite." https://sqlite.org/.

[49] "libconfig." http://www.hyperrealm.com/libconfig/, 2013.

[50] "The Opportunities and Challenges of Exascale Computing." http://science.energy.gov/~/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf.

[51] "tgt project, Linux SCSI Target Framework." http://stgt.sourceforge.net/.

[52] "Filesystem in Userspace." http://fuse.sourceforge.net/.

[53] "Montage - An Astronomical Image Mosaic Engine." http://montage.ipac.caltech.edu/docs/m101tutorial.html.

[54] L. Moreau, B. Ludäscher, I. Altintas, R. S. Barga, S. Bowers, S. Callahan, G. Chin, B. Clifford, S. Cohen, S. Cohen-Boulakia, *et al.*, "Special issue: The first provenance challenge," *Concurrency and computation: practice and experience*, vol. 20, no. 5, pp. 409–418, 2008.

[55] Y. Wang, T.-H. Ahn, Z. Li, and C. Pan, "Sipros/prorata: a versatile informatics system for quantitative community proteomics.," *Bioinformatics*, vol. 29, no. 16, 2013.

[56] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic identification of application i/o signatures from noisy server-side traces," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, (Santa Clara, CA), pp. 213–228, USENIX, 2014.