An Analysis Workflow-Aware Storage System for Multi-Core Active Flash Arrays

Hyogi Sim, Geoffroy Vallée, Youngjae Kim, Sudharshan S. Vazhkudai, Devesh Tiwari, and Ali R. Butt

Abstract—The need for novel data analysis is urgent in the face of a data deluge from modern applications. Traditional approaches to data analysis incur significant data movement costs, moving data back and forth between the storage system and the processor. Emerging Active Flash devices enable processing on the flash, where the data already resides. An array of such Active Flash devices allows us to revisit how analysis workflows interact with storage systems. By seamlessly blending together the flash storage and data analysis, 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, elevating data analyses as first-class citizens, and transforming AnalyzeThis into a potent analytics-aware appliance. To evaluate the AnalyzeThis system, we have adopted both emulation and simulation approaches. In particular, we have evaluated AnalyzeThis by implementing the AnalyzeThis storage system on top of the Active Flash Array's emulation platform. We have also implemented an event-driven AnalyzeThis simulator, called AnalyzeThisSim, which allows us to address the limitations of the emulation platform, e.g., performance impact of using multi-core SSDs. The results from our emulation and simulation platforms indicate that AnalyzeThis is a viable approach for expediting workflow execution and minimizing data movement.

Index Terms—Distributed systems, storage management, scientific data management

1 Introduction

Data analysis is often considered the fourth paradigm of scientific discovery, complementing theory, experiment, and simulation. Experimental facilities (e.g., Spallation Neutron Source [51]), observational devices (e.g., Sloan Digital Sky Survey [46], Large Synoptic Survey Telescope [27]) and high-performance computing (HPC) simulations of scientific phenomena on clusters (e.g., Titan supercomputer [55] and other Top500 machines [57]) produce hundreds of terabytes of data that need to analyzed to glean insights. The data products are often stored in central, shared repositories, supported by networked file systems (NFS) or parallel file systems (PFS) (e.g., Lustre [45] or GPFS [44]). Analyses that operate on these datasets are often I/O-intensive, and involve running a complex workflow job on a smaller cluster. The analysis workflow reads the input data from the central storage, applies a series of analytics kernels, such as statistics, reduction, clustering, feature extraction and legacy application routines, and writes the final, reduced data back to the storage system. We refer to the entire sequence of reading the input data, followed by analysis on a cluster, and writing the output as *Offline data analysis*.

Offline analysis incurs a substantial amount of redundant I/O, as it has to read the inputs from the storage system, and write the reduced results back. Reading back large data for analysis on a cluster exacerbates the I/O bandwidth bottleneck that is already acute in storage systems [22]. This is because, I/O bandwidth has traditionally been lagging behind the compute and memory subsystems, and the data production rates from simulations [56] and experimental facilities are compounding the problem further, creating a *storage wall*. Instead of an offline approach to data analysis, analyzing data *in-situ* on the storage system, where the data resides, can not only minimize data movement, but also expedite the time to solution of the analysis workflow. In this paper, we explore such an approach to data analysis.

To alleviate the I/O bottleneck, network-attached storage systems for clusters are being built with solid-state devices (SSD), resulting in either hybrid SSD/HDD systems or all flash arrays. The lack of mechanical moving parts, coupled with a superior I/O bandwidth and low latency, has made SSDs an attractive choice. We argue that SSDs are not only beneficial for expediting I/O, but also for on-the-fly data analysis. SSDs boast an increasing computational capability on the controllers, which have the potential to execute data analysis kernels in an *in-situ* fashion. In this model, the analysis is conducted near the data, instead of shipping the data to the compute cores of the analysis cluster.

In our prior work on Active Flash [6], [56], we explored the viability of offloading data analysis kernels onto the flash controllers, and analyzed the performance and energy tradeoffs of such an offload. We found that Active Flash outperformed offline analysis via a PFS for several analysis tasks. In this paper, we explore how such an active

- *H. Sim, G. Vallée, and S. Vazhkudai are with Oak Ridge National Laboratory, Oak Ridge, TN 37830. E-mail: {simh, valleegr, vazhkudaiss}@ornl.gov.*
- *Y. Kim is with Sogang University, Seoul 04107, South Korea. E-mail: youkim@sogang.ac.kr.*
- *D. Tiwari is with Northeastern University, Boston, MA 02115. E-mail: tiwari@northeastern.edu.*
- *A. Butt is with Virginia Tech, Blacksburg, VA 24061. E-mail: butta@cs.vt.edu.*

Manuscript received 14 Aug. 2017; revised 15 Feb. 2018; accepted 8 Apr. 2018. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Youngjae Kim.)
Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2865471

processing element can form the fabric of an entire storage system that is workflow-aware.

An array of such Active Flash devices allows us to 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 large-scale data analysis would be: writing data to an analysis-aware storage system automatically *triggers* a sequence of predefined analysis routines to be applied to the data.

Contributions. We propose *AnalyzeThis*, a storage system atop an array of Active Flash devices. 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 (Section 3.1). The analysis data object abstraction is overlaid on the Active Flash device, and this entity is referred to as the *Active Flash Element, AFE*. We mimic an AFE array using an emulation platform. We explore how scheduling, i.e., both data placement and workflow orchestration, can be performed within the storage, in a manner that minimizes unnecessary data movement between the AFEs, and optimizes workflow performance (Section 3.2). Moreover, we design easy-to-use file system interfaces with which the AFE array can be exposed to the user (Section 3.3). The FUSE-based file system layer enables users to read and write data, submit analysis workflow jobs, track and interact with them via a */proc-like* interface, and pose provenance queries to locate intermediate data (Section 3.4). Finally, we have developed an AnalyzeThis simulator, called AnalyzeThisSim, to study the performance impact of using a multi-core architecture in the AFEs (Section 6). This simulator is event-driven, capable of evaluating multi-core AFEs and host and AFE-level (fabric) schedulers. We argue that these concepts bring a fresh perspective to large-scale data analysis. Our results with real-world, complex data analysis workflows on AnalyzeThis, built atop an emulation-based AFE prototype, indicate that it is very viable, and can expedite workflows significantly.

1.1 Background on Active Flash

Here, we present a summary of our prior work, Active Flash, upon which the AnalyzeThis storage system is built.

Enabling Trends. First, we highlight the trends that make flash amenable for active processing.

High I/O throughput and internal bandwidth: SSDs offer high I/O throughput and internal bandwidth due to interleaving techniques over multiple channels and flash chips. This bandwidth is likely to increase with devices possessing more channels or flash chips with higher speed interfaces.

Availability of Spare Cycles on the SSD Controller. SSD controllers exhibit idle cycles on many workloads. For example, HPC workloads are bursty, with distinct compute and I/O phases. Typically, a busy short phase of I/O activity is followed by a long phase of computation [8], [22]. Further, the I/O activity recurs periodically (e.g., once every hour), and the total time spent on I/O is usually low (below 5 percent [23]). Even some enterprise workloads exhibit idle periods between their I/O bursts [28], [29]. Data ingest from experimental facilities, such as SNS [51] are based on the availability of beam time, and there are several opportunities for idle periods between user experiments, which involve careful calibration of the sample before the beam can be applied to it to collect data. Such workloads expose spare cycles available on the SSD controller, making it a suitable candidate for offloading data analysis tasks.

Multi-Core SSD Controllers. Recently marketed SSDs are equipped with fairly powerful mobile cores, and even multi-core controllers (e.g., a 4-core 780 MHz controller on the OCZ RevoDrive X2 [35]). Multi-core SSD controllers are likely to become more common place, and hence the available idle time on the SSD controllers will increase as well.

Active Flash. In our prior work o n Active Flash [6], [56], we presented an approach to perform in-situ data analysis on SSDs. We presented detailed performance and energy models for Active Flash and offline analysis via PFS, and studied their provisioning cost, performance, and energy consumption. Our modeling and simulation results indicated that Active Flash is better than the offline approach in helping to reduce both data movement, and energy consumption, while also improving the overall application performance. Interestingly, our results suggest that Active Flash can even help defray part of the capital expenditure of procuring flash devices through energy savings. We also studied hybrid analysis, involving processing on both flash and host cores, and explored when it might be suitable to offload analysis to flash. Next, our simulation of I/O-compute trade-offs demonstrated that internal scheduling may be used to allow Active Flash to perform data analysis without impact on I/O performance. To this end, we explored several internal scheduling strategies within the flash translation layer (FTL) such as analyzing while data written to the flash is still in the controller's DRAM, analyzing only during idle times (when there is no I/O due to data ingest), and combining idle time analysis with the scheduling of garbage collection (GC) to preempt GC interrupting an ongoing data analysis due to the lack of available free pages. Finally, we have demonstrated the feasibility of Active Flash through the construction of a prototype, based on the OpenSSD development platform, extending the OpenSSD FTL with data analysis functions. We have explored the offloading of several data analysis kernels, such as edge detection, finding local extrema, heartbeat detection, data compression, statistics, pattern matching, transpose, PCA, Rabin fingerprinting and k-means clustering, and found Active Flash to be very viable and cost-effective for such data analysis.

Fig. 1. *AnalyzeThis* overview. Figure shows analysis-awareness at each and every layer of AnalyzeThis.

2 AnalyzeThis Storage System

2.1 Goals

In this section, we discuss our key design principles.

Analysis-Awareness. Our main objective is to introduce analysis-aware semantics into the storage system. There is an urgent need to analyze the data in-situ, on the storage component, where the data already resides.

Reduce Data Movement. It is expected that in future, exascale data centers, the cost of data movement will rival that of the computation itself [17]. Thus, we need to minimize data movement in analysis workflows as well as across the AFEs within the storage.

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.

2.2 Overview

We envision AnalyzeThis as a smart, analytics pipeline-aware storage system atop an array of Active Flash devices (Fig. 1). The analysis workflow job is submitted to the AnalyzeThis storage system. As the input data to be processed becomes available on AnalyzeThis (from experiments, observations or simulations) the workflow that the user has submitted is applied to it. The final processed data, or any of the intermediate data is stored in AnalyzeThis, and may be retained therein, transferred to other repositories that may be available to the user (e.g., archive, PFS), or removed 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. In the future, we expect that such analysis-aware semantics will be adopted into existing PFS and NFS storage. Below is a bottom-up description of the system.

Active Flash Array. At the lowest level is the Active Flash array that is composed of discrete Active Flash devices, capable of running individual analysis kernels. Internally, an AFE has multiple processors that are capable of executing user-provided codes. We envision an array of such devices 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. The analysis abstraction, coupled with the Active Flash device (capable of processing), is referred to as the *Active Flash Element, "AFE."*

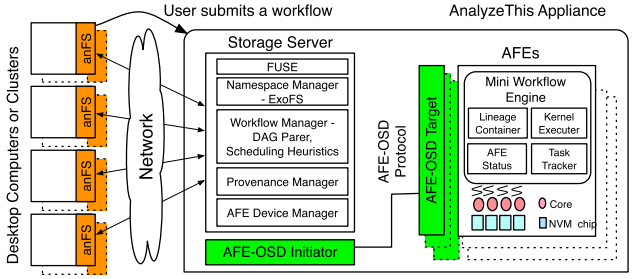
Workflow Scheduling Layer. The goal of this layer is to mimic how users interact with batch computing systems and integrate similar semantics into the storage system. Such a strategy would be a concrete step towards bridging the gap between storage and analysis workflows. Users typically submit a workflow, e.g., a PBS [16] or a DAGMAN [54] script, to a cluster'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 across the AFEs 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 AFE array, 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.

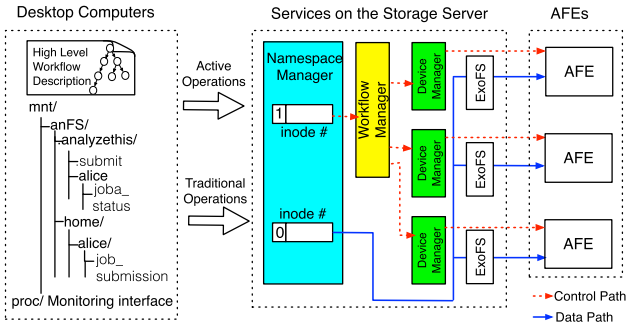
Together, these constructs provide a very potent in-situ data analytics-aware storage appliance.

3 Design and Implementation

Fig. 2a presents the architecture of AnalyzeThis. The AnalyzeThis appliance exposes a FUSE file system, *"anFS,"* to the users that can be mounted via an NFS protocol. Users submit analysis workflows and write data objects to AnalyzeThis via anFS that is mounted on their desktop computer or an analysis cluster. Thereafter, users can monitor and query the status of the jobs, and search for intermediate data products of branches of the workflow. In backend, the AnalyzeThis appliance comprises of one or more *storage servers* to which multiple Active Flash devices are connected. The storage server and the Active Flash devices run several services, and collectively help realize the analysis-aware storage appliance. Each Active Flash device runs the software



(a) AnalyzeThis Architecture: The figure shows how the desktop clients interact with AnalyzeThis by mounting the anfs client via the NFS protocol. AnalyzeThis comprises of multiple Active Flash devices, connected to one or more storage servers. Together, they run several services such as AFE, anFS servers, namespace, workflow, provenance and device managers.



(b) anFS Architecture and Data and Control Paths: The Workflow and device managers handle the active file operations or the control path. The Namespace manager, along with ExoFS, exposes the AFE as a file system to the FUSE layer for traditional data operations. The FUSE layer of anFS ties together the control and data paths into a user-level file system.

Fig. 2. AnalyzeThis architecture and components.

service that overlays an analysis object abstraction atop, transforming it into an AFE, as well as other services required to handshake with the storage server. The storage server runs services such as those required for making the AFEs available as a file system (namespace management and data protocols), distributed workflow orchestration, distributed provenance capture, and interfacing with the AFEs (device management). These services are implemented behind a FUSE layer on the storage server. Together, in a distributed fashion, they achieve workflow awareness.

Central to our design is the seamless integration of workflow scheduling and the file system. To this end, behind anFS is a *workflow scheduler* that constructs a directed acyclic graph (DAG) from the analysis workflow job. 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 storage server 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 an task, including analysis kernels, input and output datasets, and the lineage information of all the objects therein. The analysis kernels for a given workflow is assumed to be stored as a platform-dependent binary executable object (`.so` format), compiled for specific devices as needed, which can run on the AFEs.

3.1 Analysis Encapsulation

We introduce analysis awareness in the Active Flash array by building on our prior work on Active Flash that has demonstrated how to run an analysis kernel on the flash controller [6], [56]. Our goal here is to study how to overlay an *analysis object abstraction* atop the Active Flash device, both of which together form the 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 storage server 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 devise a richer construct than just files. Data formats, e.g., HDF [11], [37], [50], [59] NetCDF [24], [33], and NeXus [34], offer many desirable features such as access needs (parallel I/O, random I/O, partial I/O, etc.), portability, processing, efficient storage and self-describing behavior. However, we also need a way to tie the datasets with the analysis lifecycle in order to support future data-intensive analysis. To address this, 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, which may contain thousands of files, e.g., climate community. 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 (final or intermediate data of analysis) to retain and for how long. The analysis object abstraction transforms the dataset into an encapsulation that is more than just a pointer to a byte stream; it is now an entity that lends itself to analysis.

3.1.1 Extending OSD Implementation for AFE

We realize the analysis object abstraction using the object storage device (OSD) protocol. The OSD protocol provides a foundation to build on, by supporting storage server to AFE communication and by enabling an object container-based view of the underlying storage. However, it does not support analysis-awareness specifically. We use an open source implementation of the OSD T10 standard, Linux open-osd target [36], and extend it further with new features to implement the AFEs. We refer to our version of the OSD implementation as "AFE-OSD" (Fig. 2a). Our extensions are as follows: (i) *Mini Workflow* supports the execution of entire branches of an analysis workflow that are handed down by the higher-level Workflow Scheduler on the storage server; (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 on the storage server 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 storage server submits the mini DAG to the AFE-OSD target.

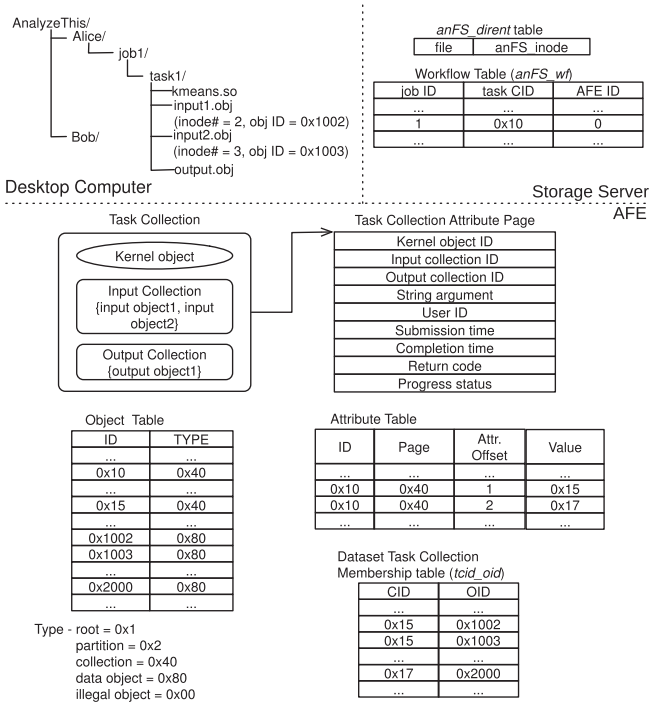


Fig. 3. Analysis object abstraction implemented using database engine. CID: task collection id, OID: object id, Attr. offset: an offset in the attribute page.

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 storage server dispatches the mini DAGs to the AFEs using a series of `ANALYZE_THIS` execution commands, with metadata on the tasks. To handle the tasks on the AFEs, we have implemented a new *analysis task collection* primitive in the AFE-OSD, which is 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 Fig. 3). Once the AFE receives the execution command, it will create an analysis task collection, and insert the task into a FIFO task queue that it maintains internally. 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 Executer: The kernel executer is a multi-threaded entity that checks the task queue and dispatches the tasks to the AFE controller. We have only used one core from the multi-core controller, but the design allows for the use of many cores. We rely on the ability of the Active Flash component of the AFE to run the kernel on the controller core, which has been studied in our prior work [6], [56]. Active Flash locates the predefined entry point (`start_kernel`) from the analysis kernel code (`.so file`), and begins the execution. In our prior work on Active Flash [6] (summarized in Section 1.1), we have explored FTL scheduling techniques to coordinate regular flash I/O, active computation and garbage collection, which can be used by the kernel executer.

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 storage server 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 storage server 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 3.2).

AFE Status. The storage server 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 storage server. The status object includes the AFE device details such as wear-out for the flash, resource usage for controller, 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 storage server as needed. Thus, the storage server 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 3.4).

Lightweight Database Infrastructure. We use a lightweight database infrastructure (Fig. 3), using SQLite [53], to implement analysis-aware semantics into the storage system. One approach is to have the storage server maintain all of the analysis workflow, data, and analysis-semantics. However, such a centralized approach is not resilient in the face of storage server crashes. Instead, we implement a decentralized approach (refer to Fig. 2a), wherein the storage server (the FUSE layer) and the AFEs (the AFE-OSD Target) maintain relevant information and linkages to collectively achieve the analysis abstraction.

The storage server 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 Fig. 3 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, the storage server 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.

3.2 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 analysis workflow script via anFS, it distinguishes this request from a normal I/O request. This is accomplished by treating the "write" request coming through the special file (.submit) as a job submission instead of normal write operation by anFS. The script is delivered to the scheduler which parses it to build a directed acyclic graph, schedules the tasks, and sends the execution requests to the AFEs via the AFE-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 analysis job's inputs are written to AnalyzeThis. This is akin to the *smart folder* concept discussed in Section 1. The analysis sequence is first registered with AnalyzeThis, and once the input data is available the tasks are executed.

Workflow Description and DAG. In our implementation, we have chosen to represent a job script using *Libconfig* [25], a widely used library for processing structured configuration files. Listing 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 by the scheduler via a two-pass process. 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.

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 [12]. We have designed and implemented several scheduling strategies that attempt to strike a balance between these two competing objectives.

Round-Robin (RR). 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 a priori information about their execution time. It picks the next available AFE in a round-robin fashion to balance the computational load. The round-robin strategy schedules the task on an available idle AFE controller, causing data movement, potentially in favor of a shorter execution time and load balance across the AFE controllers. Consequently, the technique may suffer from excessive data movement because it does not account for the amount of data to be moved.

Listing 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" ]; }
);
```

Input Locality (IL). 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. Input-locality favors a reduction in data movement to performance (execution time). In our experiments with real workflows, we observed that this scheduling policy is effective in reducing the data movement. However, it can potentially increase the overall execution time considerably because it will execute the analysis on the AFE that stores larger input, even if other AFEs are idle.

Minimum Wait (MW). To reconcile execution time and data movement, we propose to explicitly account for the data transfer time and queuing delays on the AFE controllers. The heuristic takes two inputs including a 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. Since the precise estimation of the expected wait time can be complex, we adopt a heuristic-based approach based on the data transfer time [48]. 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.

Hybrid (HY). In the hybrid strategy, we exploit the storage server within the AnalyzeThis appliance (storage server in Fig. 2a), in addition to the AFEs, to exploit additional opportunities. The storage server can run certain tasks in addition to anFS services. Running computation on the servers to which the disks are attached is a well-known practice adopted by several commercial vendors. However, hybrid processing offers more benefits by further exploiting the internal, aggregate bandwidth of the multi-channel flash device that exceeds the PCIe bandwidth between the storage server and the flash device by a factor of 2-4× [9]. AnalyzeThis does not blindly place all tasks on the AFEs or on the host storage server. The challenge is in carefully determining what to offload where (storage server versus AFE) and when. Traditional solutions that simply perform server-side processing do not address this optimization. Reduce tasks in workflows involve high data movement cost because they gather multiple intermediate inputs on the AFEs, and can be moved to the storage server. This approach has the advantage of minimizing the overhead of data movement between the AFEs, beyond what Input Locality alone can achieve, without sacrificing the parallelism. Also, tasks that cause an uneven distribution on the AFEs cause stragglers, and can be moved to the storage server (unaligned tasks). Such tasks can be identified based on profiling of the workflows. The hybrid approach can work in conjunction with any of the aforementioned scheduling techniques.

3.3 anFS File System Interface

The functionalities of AnalyzeThis are exposed to the clients via a specialized file system interface, "anFS." Since the analysis workflows operate on but do not modify the original data from scientific experiments and simulations, anFS is designed as a write-once-read-many file system. As discussed earlier (Section 3), anFS is exported to clients via NFS. Thus, operations on shared files follow the NFS consistency semantics. anFS provides standard APIs such as open (), read(), 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.

Fig. 2b 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/. FUSE provides an elegant way to develop user-level file systems. anFS provides several custom features, such as workflow execution and provenance management, which are more appropriate to be implemented in the user-space than the kernel-level. Also, a FUSE-based, user-level implementation offers better portability than a kernel-based solution. 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 3.2). The *Device Manager* provides the *control path* to the AFEs, implementing AFE-OSD (Section 3.1.1) to allow interactions with the AFEs. Finally, the *exoFS (Extended Object File System) layer* [13] provides the *data path* to the AFEs.

Namespace. anFS exposes a consolidated standard hierarchical UNIX namespace view of the files and SVFs on the AFEs. To this end, the storage server metadata table (Section 3.1.1) includes additional information associated with every stored object and information to track the AFEs on which the objects are stored (Fig. 3). 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 we employ the SQLite RDBMS [53] 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. Instead of striping, 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 Fig. 2b. 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 AFE-OSD library for further analysis and actions.

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 /procbut for workflow submission and management (Fig. 2b). 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/joblistfor 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

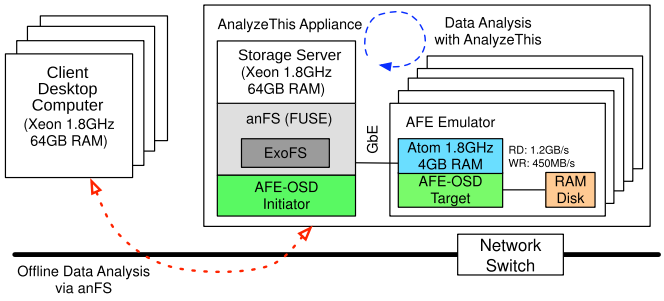


Fig. 4. AnalyzeThis testbed.

task progress, and the user can retrieve this information 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_attributecommand 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 storage server. However, this incurs added overhead on the storage server. In the future, direct PCI to PCI communication can help expedite these transfers.

Data and Workflow Security. anFS ensures data security for multiple users via the OSD2 standards. To protect the stored data, OSD maintains the ownership of objects as object attributes. When a data item is stored, it also provides the kernel-level user-id of the data owner, which is then stored in the OSD-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 the data. Similarly, when a task is scheduled on the AFE, it is associated with the user-id information, and must present these credentials to access data. The access control for the SVFs are set such that the submit SVF (.anFS/submit) is world writable, but the resulting joblist and status files are user specific. The sub-directories are named on a per-user basis, e.g., Alice's jobs are under .anFS/alice/, and the associated POSIX ACLs protect user files.

3.4 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 (Fig. 3) between the storage server (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 storage server. 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 storage server 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 storage server in the *anFS_wf* table.

TABLE 1
Workflow Input, Output and Intermediate Data Size

	Input	Intermediate	Output	Total	Object
		(MB)			(#)
Montage	51	222	153	426	113
Brain	70	155	20	245	35
Sipros	84	87	1	172	45
Grep	463	363	1	827	13

4 Experimental Setup

Testbed. Our emulation testbed (Fig. 4) is composed of the following: (1) client desktop computer that submits analysis workflows, (2) storage server within the AnalyzeThis appliance, and (3) the networked machines that emulate the AFEs (four AFEs are connected to the storage server). For the desktop computer and the storage server, we used a 1.8 GHz Intel Xeon E5-2603 processor. We emulated the AFEs using Atom machines with RAM disks, to mimic the flash controller and the internal flash chips with high I/O bandwidth to the controller. The Atom-based AFEs use a single 1.8 GHz Atom processor as the controller, a 3 GB RAM disk as the flash chip, and a 1 Gbps Ethernet connection to the storage server within the AnalyzeThis appliance. All servers run the Linux kernel 2.6.32-279. anFS offers a read and write bandwidth of 120 MB/s and 80 MB/s, respectively.

Software. AnalyzeThis has been implemented using 10 K lines of C code. We extended the OSD iSCSI target emulator from the open-osd project [36], for the AFE target. The task executions in an AFE are serialized by spawning a dedicated thread, which mimics dedicating a device controller for active processing. For the AFE-OSD driver in the storage server, we extended the OSD initiator driver in the Linux kernel. We also extended exoFS [13] to synchronize the OSD object id space with the userspace anFS. anFS has been implemented using FUSE [14], and it keeps track of metadata using SQLite [53].

Scientific Workflows. We used several real-world complex workflows. We used Montage [30], Brain Atlas [31], Sipros [60], and Grep [15] workflows. The DAG representations and the details of the workflows are shown in Fig. 6 and Table 1. The Montage workflow [30] creates a mosaic with 10 astronomy images. It uses 8 analysis kernels, and is composed of 36 tasks, several of which can be parallelized to run on the AFEs. The Brain workflow [31] creates

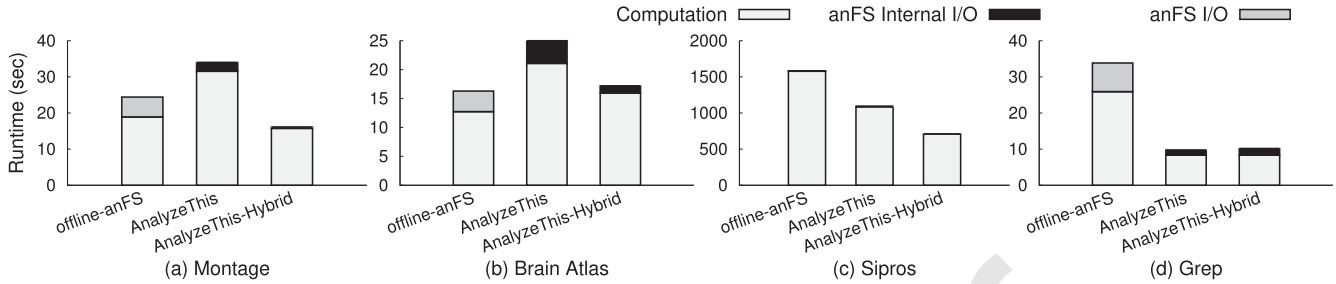


Fig. 5. Comparison of AnalyzeThis round-robin, hybrid, and offline-anFS. Multiple runs for each case, without much variance.

population-based brain atlases from the fMRI Data Center's archive of high resolution anatomical data, and is part of the first provenance challenge [31] 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. It consists of 12 analysis tasks, and uses three analysis kernels. The Grep workflow counts the occurrences of ANSI C keywords in the Linux source files.

5 Evaluating AnalyzeThis

5.1 AnalyzeThis Performance

We compare offline-anFS and AnalyzeThis. In offline-anFS, data analyses are performed on desktops or clusters by pulling data from the anFS, whereas in *AnalyzeThis*, they are performed on the AFE cores. In Fig. 5, we show the total runtime in terms of computation and I/O time. We further break down the I/O time into anFS I/O and anFS-internal I/O times (i.e., data movement between the AFEs). Therefore, a break-down of the workflow run time comprises of the following: (i) time to read the data from anFS (only offline-anFS incurs this cost), (ii) compute time of the workflow on either the desktop or the AFEs, (iii) I/O time to write the intermediate output to anFS during analysis (only for offline-anFS), (iv) data shuffling time among the AFEs (only for AnalyzeThis), and (v) time to write the final analysis output to anFS. We specifically compared the following scenarios: (i) offline analysis using one client node and anFS (offline-anFS), (ii) AnalyzeThis using four Atom-based AFEs and round-robin scheduling across the AFEs, and (iii) AnalyzeThis-hybrid using the storage server, four AFEs and round-robin across the AFEs.

In the Montage, Brain and Grep experiments for offline-anFS, the time to write the analysis outputs to anFS noticeably increases the run time (i.e., more than 20 percent of the run time is consumed by I/O operations.) while, for AnalyzeThis, the I/O time, anFS-internal I/O, is much smaller compared to the overall run time. The run time for offline-anFS for Montage and Brain is slightly lower than AnalyzeThis due to relatively less computing power on the AFEs. This demonstrates that the benefit from introducing the active computation can vary depending on the application characteristics and behavior. In addition, as AFEs begin to have multicores in the future, this small difference is likely to be overcome. In contrast, for Sipros and Grep, AnalyzeThis performs better than offline-anFS. The results indicate that offline's performance is heavily affected by the data movement costs, whereas AnalyzeThis is less impacted. Further, AnalyzeThis can free up compute resources of desktops or clusters, enabling "true" out-of-core data analysis.

Next, we evaluate (AnalyzeThis-Hybrid). For Montage, AnalyzeThis-Hybrid significantly reduced the total run time over AnalyzeThis and offline-anFS. Unaligned `mProjectPP` tasks (Fig. 6a) are executed on the storage server, which removed task stragglers. Also, more than 50 percent of data copies between AFEs are reduced by executing reduce tasks on the storage server. Similarly, for Brain, executing a single reduce task (`softmean` in Fig. 6b) on the storage server eliminated more than 75 percent of data copies, which results in a 37 percent runtime reduction compared to AnalyzeThis. Similarly, for Sipros, AnalyzeThis-hybrid is better than both AnalyzeThis and offline-anFS as it ran unaligned tasks on the storage server.

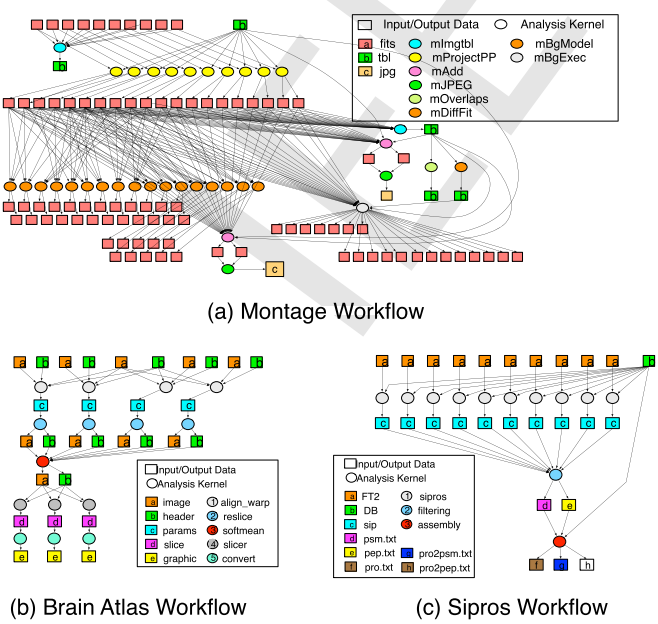
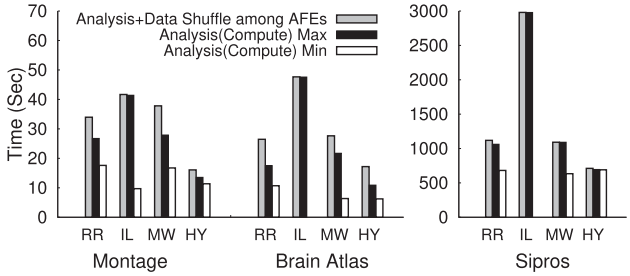


Fig. 6. The DAGs representing the workflows.

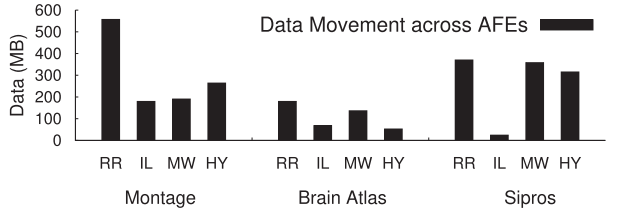
5.2 Scheduling Performance

Here, we discuss the performance of scheduling techniques.

Impact of Scheduling Heuristics. Fig. 7 compares the performance of round robin (RR), input locality (IL), minimum wait (MW), and hybrid (HY) based on AFE utilization and data movement. Fig. 7a compares the sum (first bar) of the computation time of the 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. Fig. 7b



(a) Total run-time and AFE core utilization



(b) Data movement

Fig. 7. Performance of scheduling heuristics.

shows the amount of data shuffled between the AFEs. An optimal technique strikes a balance between runtime, data movement, and AFE utilization.

HY and RR show a balanced load distribution across the AFEs with the least variability in utilization. However, RR incurs the most data movement. IL can improve runtime by significantly reducing data movement, however it may degrade the overall performance due to inefficient load distribution. IL shows higher runtimes than RR in all workflows. 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 all of the tasks; and in Sipros, only one AFE was used during analysis. HY and MW perform best in reconciling AFE utilization and data movement cost. For Montage, MW shows a 10 percent lower runtime than IL by incurring a 6 percent increase in data movement. For Brain, RR and MW show very close runtimes, but MW further reduces the data movement cost of RR by 16 percent, with less core utilization, suggesting that it is likely to be more energy-efficient. For Sipros, MW shows a 2.4 percent lower runtime than RR while reducing the data movement cost by 3 percent. By executing reduce tasks on the storage server, HY significantly reduces the runtimes over other scheduling algorithms for all workflows. In Montage and Brain, this also reduces data movement cost by 52 and 76 percent over RR, respectively.

Scaling Experiments. We performed scalability experiments for AnalyzeThis by increasing the number of AFEs. 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. For instance, in the Brain Atlas workflow, using more than four AFEs does not improve the performance further [47].

Utilizing Host Cores for Analysis. We have evaluated the impact of the host processor utilization in *AnalyzeThis*, in a Hybrid workflow run. We run the reduce tasks of the workflow on the host processor. The reduce task collects multiple intermediate inputs from the AFEs, which can result in high data movement costs between AFEs. And unaligned operations often create stragglers. Running these unaligned operations on the host processor can eliminate the runtime effects from these stragglers.

In Fig. 7, shows the results of our evaluation for RR and Hybrid scheduling algorithms with Montage, BrainAtlas, and Sipros workloads. We observe that both runtime and data copy overhead are significantly reduced by Hybrid compared with RR. In Montage, unalgined `mProjectPP` tasks were executed on the host processor, which does away with the extra parallel wave of 10 seconds. Similar improvements were observed in the Sipros results, where the extra wave is more than 300 seconds. Also, in Montage, more than 50 percent of the data copies are reduced by executing reduce tasks on the host processor. In BrainAtlas, executing a single reduce task (`softmean`) on the host eliminates more than 75 percent of the data movement cost, which results in 37 percent runtime improvement.

Provenance Performance. We have conducted the experiments of AnalyzeThis for the provenance queries using the BrainAtlas workflow, and our evaluation results are shown in our prior work [47].

6 Evaluating AnalyzeThis with Multi-Core Active Flash Elements

Recent SSDs are configured with multicore CPUs. However, in the current emulation-based approach, each single core AFE is emulated as a server. Therefore, single core AFE arrays can only be tested, and scalable experiments require expensive hardware server resources. Therefore, in this section, we describe a simulation-based study of AnalyzeThis to evaluate scalable multicore AFE arrays. We have developed an event-driven simulator, AnalyzeThisSim, which simulates workflow processing in the AnalyzeThis emulation framework (in Section 3). We assume that a set of AFEs are exclusively allocated to a single job, similarly to other resource allocation policies in the scientific computing.

6.1 AnalyzeThis Simulator

The AnalyzeThisSim takes workflow script as its input and reports timing information of each step during the workflow process. Specifically, the AnalyzeThisSim runs as follows: First, it parses an input workflow script and places all the initial input files of the workflow across the array of AFEs. AnalyzeThisSim supports two initial data placement policies of random and round-robin. Once the initial files are deployed, AnalyzeThisSim starts the execution of the workflow. AnalyzeThis parases all the tasks and files in the workflow and triggers internal events accordingly. For instance, a task event is generated based on the availability of the input files. If all the input files are available, AnalyzeThisSim schedules the task to the target core of the AFE. The event-driven design of the simulator allows calculate the time taken to send the input file, the execution time of the task, and the time to write the output file to the local AFE. All these times are calculated based on task in the workflow and hardware characteristics of the emulated target system by the simulator. By iterating over the workflow, all the tasks are progressively executed. AnalyzeThisSim collects, stores, and updates all the metrics related to job execution, including implicit data movement.

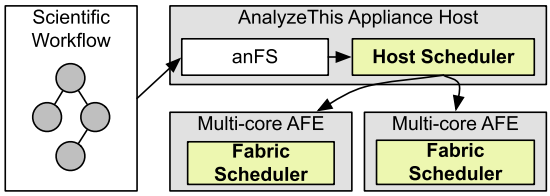


Fig. 8. Multi-level scheduling architecture in AnalyzeThis with multi-core AFEs.

```
[AFE]
    cores_per_afe    = 4
    read_bandwidth   = 550    # in MB/s
    write_bandwidth  = 450    # in MB/s
    read_latency     = 15     # in ms
    write_latency    = 15     # in ms

[SERVER]
    number_afes      = 4      # AFEs in array
```

Fig. 9. Example: simulation platform description INI File.

AnalyzeThisSim was implemented considering the following: (i) an AFE is characterized by the number of cores, read bandwidth, write bandwidth, read latency and write latency, and (ii) all AFEs are the same in terms of number of cores, bandwidth and latencies (for both read and write). A key aspect of the AnalyzeThisSim is the capability of simulating various configurations of the array of AFEs. The simulator uses INI file to describe the AFE array as shown in Fig. 9.

Validation. AnalyzeThisSim has been implemented to accurately simulate the architecture and behavior of the emulator. Moreover, we have validated the validated simulation and emulation results with Montage and BrainAtlas workloads. We recorded the timing information at important tracing points. For example, when a task is scheduled or executed in the simulation and emulation run, we have designed the simulator to exhibit less than 10 percent of errors from the tracing points in the emulation.

6.2 Multi-Level Scheduling Framework

The scheduling strategy studied in the emulator-based approach was implemented at the host-level, to manage task execution and data movement across AFEs. On the other hand, each AFE can implement its own scheduling strategy, as it is since it is composed of multiple cores capable of task execution. As shown in Fig. 8, we implement two-level schedulers, at host level and at the AFE level, which attempts AFE level, to study the efficiency of different levels of scheduling in the multicore AFE environment.

- *Host-scheduler:* The host scheduler is a host-level scheduler, in charge of executing a set of tasks on the local AFE or the array of AFEs. Since the host scheduler deals with input files for the kernels that need to be executed, it is in charge of copying and moving files to and from other AFEs, as well as starting the execution of the task on a specific AFE when all the required files are locally available.
- *Fabric Scheduler:* The fabric-scheduler is a device-level scheduler within each AFE, placing a task on the various cores of the AFE and receiving/sending information from/to the host-scheduler for scheduling purposes. In addition, we define a set of internal status records, each of which is represented as a key/value pair, i.e., the key being a unique identifier for the identification of a specific characteristic of any SSD. For instance, the write amplification (WA) ratio could be represented via the (WA, X) key/value pair where X denotes the write amplification ratio of the SSDs. Such status records are used as metrics for the scheduling policy.

6.3 Performance Impact of Using Multi-Core AFEs

To study the performance impact of using multi-core AFEs, we have used used the same set of workflows from the previous emulation-based experiments. Specifically, we ran a series of simulations and collected runtime statistics of Montage, BrainAtlas and Grep workflows. For each run, we configure a specific number of AFEs and cores per a single AFE. We vary the number of AFEs from 1 to 8, while increasing the number of cores per each AFE from 1 to 12. We have also also implemented three scheduling policies in the simulator: Round-Robin (RR), Write Amplification (WA) and Random.

The RR policy schedules tasks across AFEs. When the AFE is selected, the scheduler still schedules the tasks on the local cores based on a round-robin algorithm. A task to an AFE, the The WA policy calculates the write amplification ratio of all AFEs before assigning the task to the AFE and then it selects the AFE with the lowest ratio. To calculate the ratio, the WA algorithm implements the following model:

$$\sum Size(Input\_Files) + \frac{\sum Size(File\_Transfers)}{\sum Size(Input\_Fles)}, \quad (1)$$

where $\sum Size(Input\_Files)$ is the total size of the input files required by the task, and $\sum Size(File\_Transfers)$ is the sizes of the files that would need to be transferred to the AFE, that are not locally present. Task scheduling of cores in the AFE follows a round-robin algorithm. The Random policy randomly selects the target AFE when scheduling a task and schedules the task on the cores according to a round-robin algorithm (reusing the code in the RR policy). All files are placed on the AFEs according to the round-robin policy.

Fig. 10 shows the results of the Montage, BrainAtlas and Grep workflows using RR and WA policies. From the figure, we can observe that the overall execution of various workflows can be greatly improved by adding more AFEs or adding hardware parallelism using multi-cores on the AFE. However, due to the limited parallelism available in the workflow, it can be seen that adding more computational resources (more AFEs or more cores to the AFE) will not improve the performance. This can be explained by the fact that overall performance is limited by the number of data transfers between AFEs or the amount of data transferred. In particular, WA policy results show high performance using AFE with a large number of cores, over using a large number of AFEs with a small number of cores. The use of a multi-core AFE minimizes the data transfer overhead between AFEs.

Especially, we can see there is a slight difference in performance when comparing simulated and emulated results (Figs. 10 and 5 respectively) using four single-core AFEs

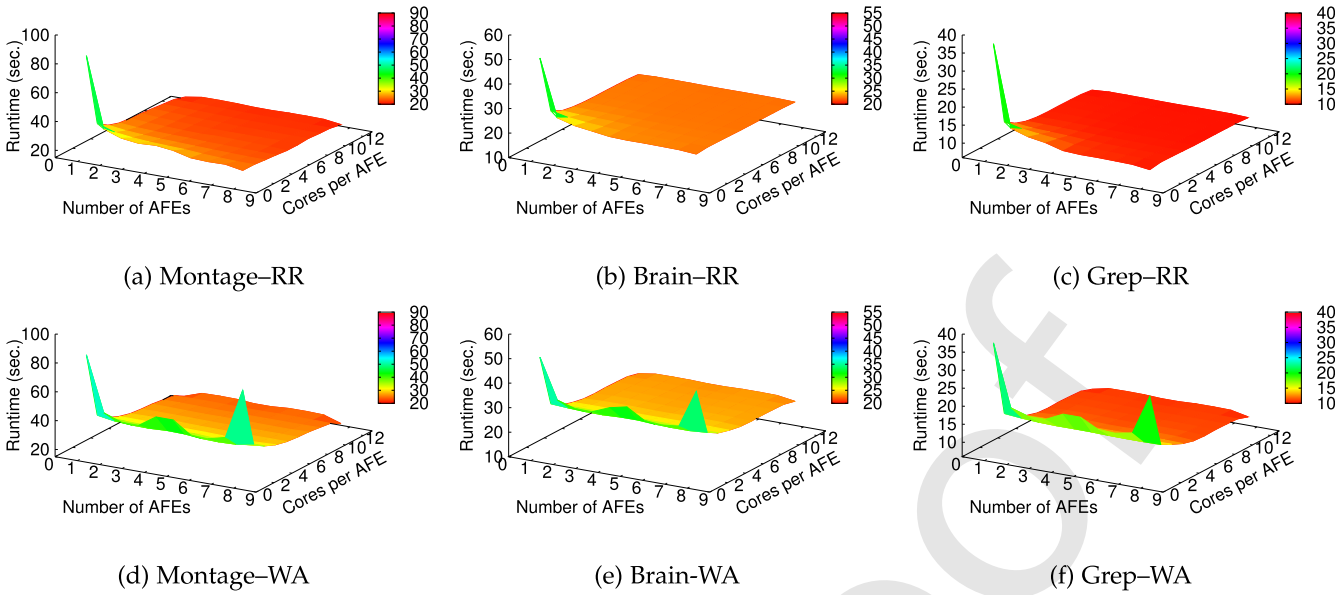


Fig. 10. Simulation results of workflow execution with Round-Robin (RR) and Write-Amplification (WA) policies.

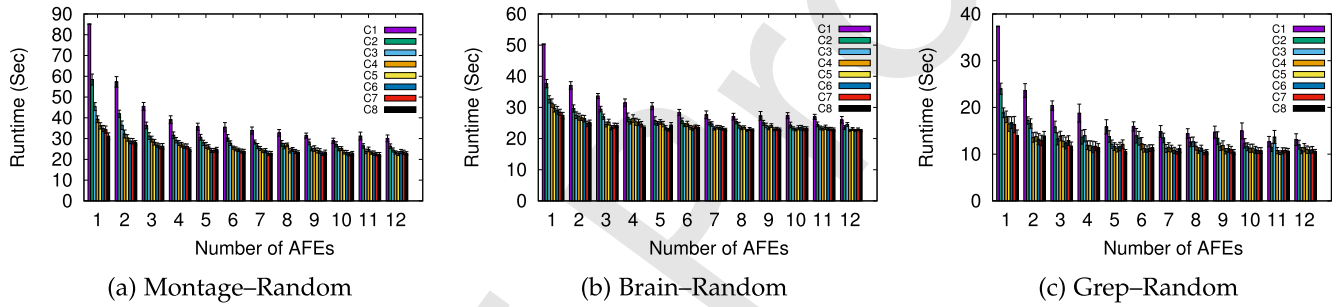


Fig. 11. Simulation results with Random policy. The 99 percent confidence intervals are shown in error bars. The number N in CN of the label represents the number of cores in the AFE.

with the RR scheduling policy. This is due to the difference in the initial file placement, i.e., in the emulator initial input files were copied to the distributed file system without control over their placement, while the simulator places files on the AFE according to the round-robin policy. We observe that the performance impact of the initial data placement varies depending on workloads. For instance, Grep workload only exhibits 3 percent performance difference (33.97 seconds in emulation and 38.48 seconds in simulation), while Montage and BrainAtlas workload shows 11 and 16 percent variation from the initial data placement.

Fig. 11, a two-dimensional graph shows the results for a random policy. We run 50 iterations per experiment and the figure show the results with an average and 99 percentile error bar graph. We can see that using a larger number of AFEs with more cores will dramatically improve the execution of various workflows. A slight variance of performance is observed in each experiment, which is not significant. In particular, in experiments with less than 4 AFEs, fewer AFEs with small number of cores can have higher workflow processing performance than many AFE approaches with fewer cores. However, due to workflow parallelism limitation, there is little difference in performance improvement in four or more AFE experiments, regardless the number of AFEs and the number of cores.

7 Related Work

Migrating tasks to disks has been explored before [20], [40]. There is a renewed interest in active processing given the recent advances in SSD technology [42]. Recent efforts, such as iSSD [9], SmartSSD [19], and Active Flash [56] have demonstrated the feasibility and the potential of processing on the SSD. In addition, recently, several studies have been introduced to build SSDs using key-value interfaces [18], [21]. 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 protocol to enable computation within a storage device. The OSD T10 standard [41], [61], [62] defines a communication protocol between the host and the OSD. Recent efforts leverage the protocol for different purposes, including executing remote kernels [41], security, and QoS [39], [62]. 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 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

TABLE 2
Comparison with Related Active Storage Systems and Worklflow-Aware Systems

Systems	Active Device	Workflow	OSD Model	Provenance	FS Interface	In-Situ
Provenance-awareness						
PASS [32], LinFS [43]	·	·	·	×	×	·
VDT [58]	·	·	·	×	·	·
Workflow·awareness						
BadFS [5]	·	×	·	·	·	·
WOSS [2]	·	×	·	·	×	·
Kepler [3]	·	×	·	×	·	·
Active Storage						
Active Disks [1], Active Flash [56], iSSD [9]	×	·	·	·	·	×
SmartSSD [19]	×	·	×	·	·	×
Data Analytics Appliance						
IBM Netezza [49]	·	·	·	×	·	×
Active Computation in PFS						
PVFS [52], Lustre [38]	·	·	×	·	×	×
I/O Middleware						
ADIOS [26]	·	×	·	·	·	×
AnalyzeThis	×	×	×	×	×	×

'×' means that a system implements the listed feature, whereas '·' implies that the system does not provide the feature.

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.

Some extensions to parallel file systems, e.g., PVFS [52] and Lustre [38], provide support for analysis on the I/O node's computing core. However, they are not workflow-aware, a key trait for efficient analysis execution, and neither is the analysis conducted on the storage device. The ADIOS [26] I/O middleware uses a subset of staging nodes alongside a running simulation on a cluster to reduce the simulation output on-the-fly; while workflow-aware, it also only uses the computing elements of the staging nodes. Instead, AnalyzeThis uses the AFEs on the storage themselves, obviating the need for a separate set of staging nodes for analysis. Enterprise solutions such as IBM Netezza [49] enable provenance tracking and in-situ analysis, but lack an easy-to-use file system interface and workflow-awareness. Workflow- and provenance-aware systems, such as PASS [32], LinFS [43], BadFS [5], WOSS [2], and Kepler [3], are not meant for in-situ analysis. The batch-aware distributed file system (BadFS) attempts to orchestrate IO-intensive batch workloads and data movement on remote systems, by layering a scheduler atop storage and compute systems in a grid network. In contrast, AnalyzeThis operates on AFEs, scheduling and colocating data and computation therein. Compared to dedicated provenance systems like PASS and LinFS, lineage tracking in AnalyzeThis is a natural byproduct of executing workflows in the storage. Distributed execution engines, such as Dryad [63], Nephele [4], Hyracks [7], and MapReduce [10], can execute data-intensive DAG-based workflows on distributed computing resources. AnalyzeThis fundamentally differs from these systems as it exploits the SSDs as the primary computing resources.

8 Conclusion

The need to facilitate efficient data analysis is crucial to derive insights from mountains of data. However, extant techniques incur excessive data movement on the storage system. 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 array, 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. We have developed the AnalyzeThis storage system on top of an emulation platform of the Active Flash array. In addition, We have developed an event-driven AnalyzeThis simulator to evaluate the highly scalable AnalyzeThis environment with multicore AFEs. Our evaluation of AnalyzeThis shows that is viable, and can be used to capture complex workflows. In future, we plan to investigate a distributed appliance model, with multiple servers, each with its own AFE arrays, thereby introducing analysis-awareness semantics into a distributed file system.

Acknowledgments

This research was supported in part by the U.S. DOE's Office of Advanced Scientific Computing Research (ASCR) under the Scientific data management program, by NSF through grants CNS-1405697 and CNS-1422788, and Next-Generation Information Computing Development Program through National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7080243). The work was also supported by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE, under the contract No. DE-AC05-00OR22725.

References

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

[2] 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:1301.6195*, 2013.

[3] 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 *Proc. 16th Int. Conf. Sci. Statist. Database Manage.*, 2004, pp. 423–424.

[4] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/PACTs: A programming model and execution framework for web-scale analytical processing," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 119–130.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Explicit control a batch-aware distributed file system," in *Proc. 1st Conf. Symp. Networked Syst. Des. Implementation - Vol. 1*, 2004, pp. 27–27.

[6] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman, "Active flash: Out-of-core data analytics on flash storage," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.

[7] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 1151–1162.

[8] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *Trans. Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011.

[9] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent SSDs," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, 2013, pp. 91–102.

[10] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation - Vol. 6*, 2004, pp. 10–10.

[11] HDF5 - A New Generation of HDF. [Online]. Available: http://hdf.ncsa.uiuc.edu/HDF5/doc/

[12] The Opportunities and Challenges of Exascale Computing. [Online]. Available: http://science.energy.gov/ /media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf

[13] exofs [LWN.net]. [Online]. Available: http://lwn.net/Articles/318564/

[14] Filesystem in Userspace. [Online]. Available: http://fuse.sourceforge.net/

[15] Grep - Hadoop Wiki. [Online]. Available: http://wiki.apache.org/hadoop/Grep.

[16] R. L. Henderson, "Job scheduling under the portable batch system," in *Job Scheduling Strategies for Parallel Processing*. New York, NY, USA: Springer, 1995, pp. 279–294.

[17] DOE Exascale Initiative Technical RoadMap, 2009. [Online].Available: http://extremecomputing.labworks.org/hardware/collaboration/EI-RoadMapV21-SanDiego.pdf

[18] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swason, "KAML: A flexible, high-performance key-value SSD," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 373–384.

[19] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol.*, 2013, pp. 1–12.

[20] 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.

[21] Samsung key value ssd enables high performance scaling. [Online]. Available: https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Key_Value_Technology_Brief_v7.pdf

[22] Y. Kim, R. Gunasekaran, G. Shipman, D. Dillow, Z. Zhang, and B. Settlemyer, "Workload characterization of a leadership class storage cluster," in *Proc. 15th Petascale Data Storage Workshop*, Nov. 2010, pp. 1–5.

[23] Computational Science Requirements for Leadership Computing, 2007. [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2010/03/ORNL_TM-2007_44.pdf

[24] 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 *Proc. SC2003: High Perform. Netw. Comput.*, 2003.

[25] libconfig, 2013. [Online]. Available: http://www.hyperrealm.com/libconfig/

[26] 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 *Proc. 6th Int. Workshop Challenges Large Appl. Distrib. Environments*, 2008, pp. 15–24.

[27] The New Sky | LSST. [Online]. Available: http://www.lsst.org/lsst/

[28] N. Mi, A. Riska, E. Smirni, and E. Riedel, "Enhancing data availability in disk drives through background activities," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC*, Jun. 2008, pp. 492–501.

[29] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel, "Efficient management of idleness in storage systems," *Trans. Storage*, vol. 5, no. 2, pp. 4:1–4:25, Jun. 2009.

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

[31] 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 Comput.: Practice Experience*, vol. 20, no. 5, pp. 409–418, 2008.

[32] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proc. Annu. Conf. USENIX '06 Annu. Tech. Conf.*, 2006, pp. 4–4.

[33] NetCDF Documentation. [Online]. Available: http://www.unidata.ucar.edu/packages/netcdf/docs.html

[34] Nexus. [Online]. Available: http://trac.nexusformat.org/code/wiki

[35] OCZ RevoDrive 3 X2 (EOL) PCI Express (PCIe) SSD. [Online]. Available: http://ocz.com/consumer/revodrive-3-x2-pcie-ssd

[36] Open-OSD project, 2013. [Online]. Available: http://www.open-osd.org

[37] HDF5 Tutorial: Parallel HDF5 Topics. [Online]. Available: http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/parallel.html

[38] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *Proc. ACM/IEEE Conf. Supercomputing*, 2007, Art. no. 28.

[39] L. Qin and D. Feng, "Active storage framework for object-based storage device," in *Proc. 20th Int. Conf. Adv. Inf. Netw. Appl.*, 2006, pp. 97–101.

[40] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large scale data mining and multimedia applications," in *Proc. 24th Conf. Very Large Databases*, 1998, pp. 62–73.

[41] M. T. Runde, W. G. Stevens, P. A. Wortman, and J. A. Chandy, "An active storage framework for object storage devices," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.

[42] Samsung SSD. [Online]. Available: http://www.samsung.com/uk/consumer/memory-cards-hdd-odd/ssd/830

[43] C. Sar and P. Cao, Lineage File System, 2005. [Online]. Available: http://crypto. stanford. edu/cao/lineage.html

[44] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, Art. no. 19.

[45] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proc. Linux Symp.*, 2003.

[46] SDSS-III DR12. [Online]. Available: http://www.sdss.org

[47] H. Sim, Y. Kim, S. S. Vazhkudai, D. Tiwari, A. Anwar, A. R. Butt, and L. Ramakrishnan, "AnalyzeThis: An analysis workflow-aware storage system," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.

[48] A. Simonet, G. Fedak, M. Ripeanu, and S. Al-Kiswany , "Active data: A data-centric approach to data life-cycle management," in *Proc. 8th Parallel Data Storage Workshop*, 2013, pp. 37–44.

[49] M. Singh and B. Leonhardi, "Introduction to the IBM Netezza warehouse appliance," in *Proc. Conf. Center Adv. Stud. Collaborative Res.*, 2011, pp. 385–386.

[50] HDF 4.1r3 User's Guide. [Online]. Available: http://hdf.ncsa.uiuc.edu/UG41r3_html/

[51] Spallation Neutron Source | ORNL Neutron Sciences. [Online]. Available: http://neutrons.ornl.gov/facilities/SNS/

[52] 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 *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–12.

[53] SQLite. [Online]. Available: https://sqlite.org/

[54] DAGMan: A Directed Acyclic Graph Manager. [Online]. Available: http://research.cs.wisc.edu/htcondor/dagman/dagman.html

[55] Introducing Titan. [Online]. Available: https://www.olcf.ornl.gov/titan/

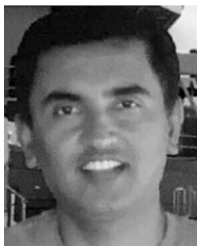
[56] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 119–132.

[57] Top 500 Supercomputer Sites. [Online]. Available: http://www.top500.org/
[58] Virtual Data Toolkit. [Online]. Available: http://vdt.cs.wisc.edu/
[59] G. Velamparampil, "Data management techniques to handle large data arrays in HDF," Master's thesis, Dept. Comput. Sci., Univ. Illinois, Champaign, IL, USA, Jan. 1997.
[60] Y. Wang, T.-H. Ahn, Z. Li, and C. Pan, "Sipros/ProRata: A versatile informatics system for quantitative community proteomics," *Bioinf.*, vol. 29, no. 16, pp. 2064–2065, 2013.
[61] R. O. Weber, "Information technology - SCSI object-based storage device commands (OSD)," *Tech. Council Proposal Document*, vol. 10, pp. 201–225, 2004.
[62] 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 *Proc. IEEE 27th Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–12.
[63] 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 *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 1–14.

Hyogi Sim received the BS degree in civil engineering and the MS degree in computer engineering from Hanyang University in South Korea, and the MS degree in computer science from Virginia Tech, in 2014. He is currently working towards the PhD degree at Virginia Tech. He joined Oak Ridge National Laboratory, in 2015, as a post-masters associate. During this appointment, he conducted research and development on active storage systems and scientific data management for HPC systems. He is currently an HPC systems engineer with Oak Ridge National Laboratory. His primary role is to design and develop a checkpoint-restart storage system for the exascale computing project. His areas of interest include storage systems and distributed systems.

Geoffroy Vallée received the MS degree in computer science from the Universite de Versailles Saint-Quentin-en-Yvelines, France, in 2000, the PhD degree in computer science from Universite Rennes 1, France, in 2004 during which he collaborated with both INRIA and Electricite de France (EDF). He joined Oak Ridge National Laboratory in 2004 as a postdoctoral researcher and became a research scientist in 2007. His research interests include system for high-performance computing, including operating systems, networking substrates, run-time systems, resilience, and fault tolerance.

Youngjae Kim received the BS degree in computer science from Sogang University, Republic of Korea, in 2001, and the MS degree from KAIST, in 2003, and the PhD degree in computer science and engineering from Pennsylvania State University, University Park, Pennsylvania, in 2009. He is currently an assistant professor with the Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea. Before joining Sogang University, he was a staff scientist with the US Department of Energy's Oak Ridge National Laboratory (2009-2015) and an assistant professor with Ajou University, Suwon, Republic of Korea (2015-2016). His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.



Sudharshan S. Vazhkudai received the master's and PhD degrees in computer science from the University of Mississippi, in 2003 and 1998, respectively. He leads the Technology Integration (TechInt) group in the National Center for Computational Sciences (NCCS), Oak Ridge National Laboratory (ORNL). NCCS hosts the Oak Ridge Leadership Computing Facility (OLCF), which is home to the 27 petaflops Titan supercomputer. He leads a group of 17 HPC researchers and systems software engineers; the group is charged with delivering new technologies into OLCF by identifying gaps in the system software/hardware stack, and developing, hardening and deploying solutions. His group's technology scope includes the deep-storage hierarchy, non-volatile memory, system architecture, system monitoring, and data and metadata management.

Devesh Tiwari received the BS degree in computer science and engineering from the Indian Institute of Technology (IIT) Kanpur, India, and the PhD degree in electrical and computer engineering from North Carolina State University. He is an assistant professor with Northeastern University. Before joining Northeastern, Devesh was a staff scientist with the Oak Ridge National Laboratory. His research interests include designing efficient and scalable large scale computing and storage systems. His research publications have received best paper award nominations at conferences including Supercomputing (SC), Dependable Systems and Networks (DSN), and Parallel & Distributed Processing Symposium (IPDPS). His work has appeared in various conferences such as USENIX FAST, SC, DSN, HPCA, MICRO, IPDPS, and have been covered by the news media including Slashdot and HPCWire.

Ali R. Butt received the PhD degree in electrical and computer engineering from Purdue University, in 2006. He is a recipient of an NSF CAREER Award (2008), IBM Faculty Awards (2008, 2015), a VT College of Engineering (COE) Dean's award for "Outstanding New Assistant Professor" (2009), an IBM Shared University Research Award (2009), and NetApp Faculty Fellowships (2011, 2015). He was named a VT COE Faculty Fellow in 2013. He was an academic visitor at IBM Almaden Research Center (Summer 2012) and a visiting research fellow at Queen's University of Belfast (Summer 2013). He has served as an associate editor for the *ACM Transactions on Storage* (2016-present), the *IEEE Transactions on Parallel and Distributed Systems* (2013-present), *Cluster Computing: The Journal of Networks, Software Tools and Applications* (2013-present), and the *Sustainable Computing: Informatics and Systems* (2010-2015). He is an alumni of the National Academy of Engineering's US Frontiers of Engineering (FOE) Symposium (2009), US-Japan FOE (2012), and National Academy of Science's AA Symposium on Sensor Science (2015). He was also an organizer for the US FOE in 2010. His research interests include distributed computing systems, cloud computing, file and storage systems, Internet of Things, I/O systems, and operating systems. At Virginia Tech he leads the Distributed Systems & Storage Laboratory (DSSL).

Queries to the Author

Q1. Please provide complete bibliography details for Ref. [2].
Q2. Please provide publication year for Refs. [11], [12], [13], [14], [15], [21], [27], [30], [33], [34], [35], [37], [42], [46], [50], [53], [57], and [58].
Q3. Please provide page-range for Refs. [24], and [45].