ϕSched: A Heterogeneity-Aware Hadoop Workflow Scheduler

Krish K.R., Ali Anwar, Ali R. Butt
Department of Computer Science, Virginia Tech
Email: {kris, ali, butta}@cs.vt.edu

***Abstract*—Enterprise Hadoop applications now routinely comprise complex workflows that are managed by specialized workflow schedulers such as Oozie. The resources are assumed to be similar or homogeneous and data locality is often the only scheduling constraint considered. However, introduction of specialized architectures and regular system upgrades lead to Hadoop data center hardware becoming increasingly heterogeneous, in that a data center may have several clusters each boasting different characteristics. However, the workflow scheduler is not aware of such heterogeneity, and thus cannot ensure that a cluster selected based on data locality is also suitable for supporting the jobs efficiently in terms of execution time and resource consumption.**

In this paper, we adopt a quantitative approach where we first study detailed behavior of various representative Hadoop applications running on four different hardware configurations. Next, we incorporate this information into a hardware-aware scheduler, ϕSched, to improve the resource–application match. To ensure that job associated data is available locally (or nearby) to a cluster in a multi-cluster deployment, we configure a single Hadoop Distributed File System (HDFS) instance across all the participating clusters. We also design and implement region-aware data placement and retrieval for HDFS in order to reduce the network overhead and achieve cluster-level data locality.

We evaluate our approach using experiments on Amazon EC2 with four clusters of eight homogeneous nodes each, where each cluster has a different hardware configuration. We find that ϕSched's optimized placement of applications across the test clusters reduces the execution time of the test applications by 18.7%, on average, when compared to extant hardware oblivious scheduling. Moreover, our HDFS enhancement increases the I/O throughput by up to 23% and the average I/O rate by up to 26% for the *TestDFSIO* benchmark.

I. Introduction

MapReduce [12] and Hadoop [3] have become synonymous with big-data frameworks for supporting a variety of applications [7], [14], [24]. Hadoop deployments now regularly boast a range of hardware from massive-core machines to low-power ARM-based devices [10], [32]. The setups are becoming heterogeneous, both from the use of advance hardware technologies and due to regular upgrades to the system. This in effect leads to a Hadoop deployment resembling a cluster of clusters that each has distinct hardware characteristics. In this paper, our goal is to sustain Hadoop in the face of such underlying heterogeneous hardware.

Hadoop applications are also becoming more intricate, and now comprise complex workflows with a large number of iterative jobs, interactive querying, as well as traditional batch-friendly long running tasks [9]. Moreover, the

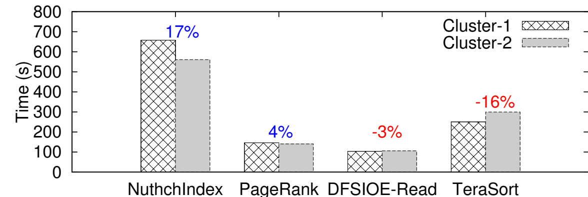


Fig. 1. Observed variation in application execution time on two different hardware configurations. The percentages represent the variation in execution time for *Cluster-1* vs. *Cluster-2*. *Cluster-1* is 8 nodes, 6.5 ECUs, 17.1 GB ram, 420 GB SATA HDD; *Cluster-2* is 8 nodes, 6.5 ECUs, 7.5 GB ram, 32 GB SATA SSD.

workflows are realized through a variety of high-level tools and languages [18] instead of manual MapReduce programming. Therefore, systems such as Oozie [20], Nova [27], and Hadoop+Kepler [33] have been developed to manage and schedule the workflows, and provide ease of use. The main goals of the workflow schedulers are to support high scalability, multi-tenancy, security, and inter operability [20]. The challenge is that extant workflow schedulers are (mostly) oblivious of the underlying hardware architecture. Thus, the schedulers do not consider in their scheduling decisions the varying execution characteristics such as CPU, memory, storage, and network usage of Hadoop applications on heterogeneous computing substrates that are quickly becoming the norm.

To further study the problem, we ran four MapReduce applications[1] namely *NuthchIndex*, *PageRank*, *DFSIOE-Read*, and *TeraSort*, on two different clusters configurations. Figure 1 shows the observed execution time for the studied cases. For *NuthchIndex* and *PageRank*, *Cluster-1* performed 17% and 4% faster than *Cluster-2*, respectively. In contrast, for *DFSIOE-Read* and *TeraSort*, *Cluster-1* performs 3% and 16% slower, respectively. In a deployment where *Cluster-1* and *Cluster-2* may exist together, an ideal schedule would place *NuthchIndex* and *PageRank* on *Cluster-2*, and *DFSIOE-Read* and *TeraSort* on *Cluster-1*. This yields an execution time improvement of 2% (*DFSIOE-Read*) to 20% (*TeraSort*) compared to the worst case placement (*Cluster-1* with *NuthchIndex* and *PageRank*; *Cluster-2* with *DFSIOE-Read* and *TeraSort*). Thus, it is crucial to consider hardware-specific application performance when scheduling jobs on heterogeneous clusters.

[1]We selected the applications to especially highlight the range of impact that different configurations can have.

In this paper, we propose to consider applications behavior on specific hardware configurations when scheduling Hadoop workflows. We assume that a deployment is made of one or more *resource clusters* each with a different hardware configuration, and that the resources within a cluster are similar/homogeneous. For this work, we focus on variations in performance characteristics, where the same application binaries can be run on the different clusters. However, the techniques presented here can also be extended to clusters comprising advanced architectures such as GPUs, accelerators, and Microservers. We first study characteristics such as CPU, memory, storage, and network usage for a range of representative Hadoop applications on four different hardware configurations. Next, based on our understanding of the applications, we design a hardware-heterogeneity-aware workflow scheduler, ϕSched[2], which: i) profiles applications execution on different clusters and performs statistical analysis to determine a suitable resource–application match; and ii) effectively utilizes the matching information to schedule future jobs on clusters that will yield the highest performance. Such profiling is feasible as recent research [9], [25] has shown the workflows to have very predictable characteristics, and the number of different kinds of jobs to be less than ten. To schedule a job, ϕSched examines the current utilization of the clusters and the suitability of clusters to support the job based on prior profiling. Based on these factors, ϕSched then suggests the best cluster to execute the job.

As stated above, we treat a Hadoop deployment to consist of multiple separate clusters to handle resource heterogeneity. This leads to the problem that the best cluster, C_B, to run an application in terms of execution time may not have the data associated with the application, entailing data copying/movement to C_B from the cluster, C_D, that has the data. C_D may not be able to support the application due to hardware constraints. Moreover, the data movement may be very expensive and negate the performance gain that can be realized by running the job on C_B. A similar problem is faced in standard Hadoop deployments in large Enterprises as well. For instance, Yahoo! has numerous "common data sets" that are actually stored across independently-managed storage substrates [30], i.e., the data that may be required across multiple clusters is managed and stored at only one cluster that cannot always run the jobs associated with the data. The extant solution is to use the *distcp* [5] tool to copy data from one cluster to another. However, this is very expensive, and not desirable.

Configuring a single Hadoop Distributed File System (HDFS) for all the Hadoop clusters can help mitigate the above problems. This can lead to two issues. First, a single HDFS instance may not scale to accommodate all the nodes from the multiple clusters. This is resolved by the use of HDFS Federation [29] that supports multiple master components, which ensure increased horizontal scalability. Second, the data placement supported by HDFS is not suitable to the multi-cluster setup. To this end, we enhance the HDFS with the notion of a "region," and the storage substrate attached to each cluster is associated with a unique region. We then exploit the region information to achieve better cluster locality for the data. Moreover, we also provide APIs to move data across regions and use region-specific replication factors for data items. ϕSched can leverage these APIs to extract file specific storage information such as regions in which a file is stored and the number of replicas of a file in a region. This also leads to better management of data movement when needed, e.g., by pre-staging data from one cluster to another to improve performance.

Specically, this paper makes the following contributions:

- Design a workflow management system to effectively manage multiple heterogeneous clusters.
- Develop an effective mechanism to track, record and analyze applications behavior on clusters with different hardware configurations.
- Optimize the scheduler to launch Hadoop applications on suitable clusters based on runtime analysis of prior jobs of the same kind.
- Realize enhancements for HDFS to support data-sharing between multiple clusters.

We evaluate our approach using experiments on Amazon EC2 [1] with four clusters with different hardware configurations, where each cluster had eight homogeneous nodes. Our evaluation of ϕSched reveals that the performance of application varies significantly across different hardware configurations. Experiments suggest that the hardware-aware scheduling can perform 34% faster than hardware oblivious scheduling for the studied applications. We find that ϕSched's optimized placement of applications across the test clusters reduces the execution time of the test applications by 18.7%, on average, when compared to extant hardware oblivious scheduling. Moreover, our HDFS enhancement increases the I/O throughput by up to 23% and the average I/O rate by up to 26%, for the well-known *TestDFSIO* HDFS benchmark.

II. Background

Hadoop offers an open-source implementation of the MapReduce framework that provides machine-independent programming at scale. A Hadoop cluster node consists of both compute processors and directly-attached storage. A small number of nodes (typically $12 - 24$ [6]) are grouped together and connected with a network switch to form a rack. One or more racks form the Hadoop cluster. Hadoop provides a *JobTracker* component that accepts jobs from the users and also manages the compute nodes that each run a *TaskTracker*. All data in MapReduce is represented as key-value pairs [36]. Programmers specify user defined map and reduce functions, which operate on the key-value pairs. Each TaskTracker has one or more map and reduce slots, and applications will have tens of hundreds of map and reduce tasks running on these slots. In case of heterogeneous clusters, the map/reduce tasks executing on the slowest node will

[2]The ϕ in ϕSched is inspired by the use of ϕ as the work function in solid state physics.

determine the execution time of the application [2]. Although speculative execution [39] can reduce this dependency, it leads to significant resource wastage due to re-execution of tasks.

The data management is provided by the Hadoop Distributed File System (HDFS). The main functions of HDFS are to ensure that tasks are provided with the needed data, and to protect against data loss due to failures. HDFS uses a *NameNode* component to manage worker components called *DataNodes* running on each Hadoop node. Typically, each MapReduce cluster is configured with one instance of HDFS, and the data in one cluster is not accessible (directly) from other clusters. HDFS divides all stored files into fixed-size blocks (chunks) and distributes them across DataNodes in the cluster. Moreover, the system typically maintains three replicas of each data block, two placed within the same rack and one on a different rack. The replica placement policy distributes each data block across multiple racks to ensure fault tolerance against node and rack failure. For data retrieval, a list of DataNodes ordered with respect to network proximity to the application instance is obtained from the NameNode and the nearest replicas are used. For large clusters, the default placement policy may not be efficient as it can choose two neighboring racks to store a block, while a TaskTracker that needs the block may be several network hops away.

Workflows have become an integral part of modern Hadoop applications, and are managed by workflow managers such as Apache Oozie [20] and Nova [27]. A typical workflow scheduler provides a command-line program for submitting a job that is then transformed to a control dependency Directed Acyclic Graph (DAG). The workflow scheduler is responsible for co-ordinating the various events/tasks in the DAG and allocating the events within a workflow to Hadoop. The actual execution of the tasks is done by the Hadoop's scheduler. In a multi-cluster setup, current workflow managers schedule jobs based on resource availability in a cluster as well as on completion of other dependent events or tasks, but the characteristics of the underlying hardware are not explicitly considered.

III. Design

In this section, we present the design of ϕSched and how we enhance HDFS to integrate data from multiple clusters into a single storage component.

A. Architecture Overview

Figure 2 shows the overall architecture of ϕSched. The target environment consists of multiple heterogeneous clusters, where each cluster comprise of homogeneous resources. To effectively manage the heterogeneity, we propose a hierarchical approach where we manage each cluster separately using an instance of the JobTracker, and then build a software layer for the multiple JobTrackers to interact with each other. Moreover, to avoid data partitioning between clusters, we utilize a single NameNode — that we enhance and make heterogeneity aware — to manage all the clusters.

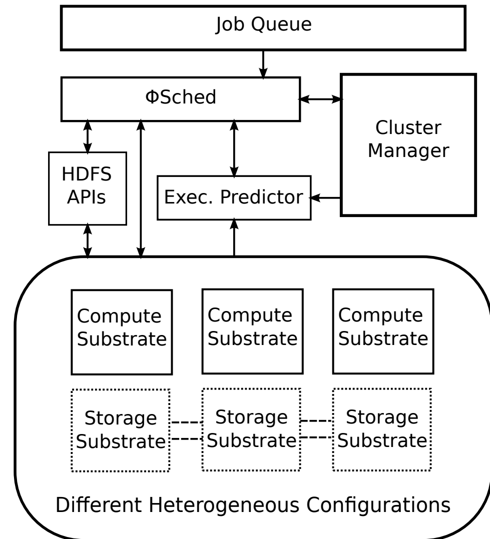


Fig. 2. ϕSched architecture overview.

The system works as follows. When a job is submitted to ϕSched, it is placed in a job queue. Next, we utilize enhanced HDFS APIs to determine the clusters where the data associated with the job is stored. We examine the current cluster load along with data availability information to schedule the job to an appropriate cluster. The actual execution is done by handing over the job to the JobTracker of the selected cluster. We also perform static statistical analysis to determine the expected execution time and resources required by the job. Moreover, the actual execution time of the job is compared with the expected values and also recorded for further fine tuning and analysis, which can then be used to guide future jobs.

B. Cluster Manager

We employ a *Cluster Manager* that manages all the clusters in a deployment. The manager tracks the load as it is assigned to the clusters, and is also responsible for profiling and predicting the utilization of resources such as CPU, memory, network and disk, for the clusters. To this end, we use both static analysis and dynamic profiling.

To drive our static analysis, we studied 12 representative MapReduce applications from HiBench [19], which cover a wide range of workload behavior such as batch processing, iterative jobs and interactive querying. This is motivated by previous research [9], [25] that has shown that MapReduce workloads are predictable in terms of their behavior and that the number of different kinds of jobs is small. By studying a range of test applications on the target clusters, we can build knowledge to better guide initial scheduling of jobs in a multi-cluster deployment.

Once a job is scheduled, the Cluster Manager switches to the dynamic profiling phase. Here, we exploit the observation that the resource consumption per task is similar across the many map (or reduce) tasks of an application, provided the underlying hardware is homogeneous. Thus, within a cluster, we can profile a single map/reduce task to predict the overall consumption, R_r, by the job. The expected R_r depends on

```
initialization : R_a = SystemResource;
foreach job j in already scheduled job list do
    R_a = R_a - R_r(j);
    wait(expectedExecutionTime of j);
    R_a = R_a + R_r(j);
end
```

Algorithm 1: Determining resource availability. R_a is the resource available in the cluster and R_r is the resource required by a job.

```
forall job j in job queue do
    forall clusters c_i in cluster list do
        History_{j,c_i}=FetchCatalog(j, c_i);
        R_r(j)=History_lookup Resource(datasize);
        //From catalog
        R_r(j)=ScaleToDevice(datasize); //From static
        analysis
        R_a = GetR_a(c_i); //From Cluster Manager
        if R_r < R_a then
            E_t(j) = History_lookup Time(j, datasize, c_i);
            OptimalList_j.add(c_i, E_t(j));
        end
    end
    Sort(OptimalList_j(c_i, E_t(j)));
end
```

Algorithm 2: Steps taken by the Execution Predictor.

the number of map and reduce tasks therein, the utilization of a single map/reduce task, as well as the input data size. Algorithm 1 shows the steps taken by the Cluster Manager to update the available resources, R_a, at a cluster as jobs are submitted and complete.

C. Execution Predictor

The main task of the *Execution Predictor* is to determine expected R_r for submitted jobs and identify suitable clusters to execute the jobs. This component maintains a catalog of job execution histories for each cluster, which includes information such as a list of recently executed applications, associated execution times, input data size, and the average R_r across prior runs of an application. The Predictor interacts with the Cluster Manager to analyze the catalog in conjunction with the R_a information, and creates lists of potential clusters that can efficiently support each application in the job queue. Moreover, the lists are sorted from least to most suitable cluster for supporting the associated job in terms of execution time and resource utilization. Algorithm 2 shows the steps taken by the Execution Predictor.

D. HDFS Enhancement

A key challenge that we face in ϕSched is to ensure that data is seamlessly available in all the clusters managed by separate JobTrackers. The solution that we employ is to run one instance of HDFS, i.e., one NameNode, to manage all the nodes across all the clusters. This leads to the problem that the default replica placement may store data associated with a job in racks that are multiple hops away from a suitable cluster for running the job. Consequently, resulting in expensive cross-cluster accesses, as well as network contention between data movement and other Hadoop operations, e.g., shuffle traffic.

To mitigate the above issue, we enhance HDFS to logically arrange participating HDFS nodes by associating each node's storage within a virtual storage group referred to as "region." Typically, all the storage in a cluster will be assigned to the same region, and storage from different clusters will be associated with different unique regions. To achieve this, we modify the DataNode to also include a region identifier as part of its characteristics specification. At the time of cluster configuration the administrator specifies the regions for the DataNodes. We also modify the NameNode to use region identifiers to group the DataNodes into their associated region. We exploit the region information to strategize when and where to place replicas of a block.

We also provide runtime APIs, shown in Table I, to manage region-aware data placement. The APIs allow the system to move files between regions, create a replica of an already existing file in a specified region and delete a file from a specified region. We note that, similarly as in default HDFS, all the APIs modify data placement at the granularity of a file and do not support block-level modifications.

Our placement policy maintains the invariant that a region contains all blocks belonging to a file. This is to avoid the inter-region fetch that might be needed for the blocks that are not in the region. Moreover, a region can have more than one replica of a file, and a file can be replicated in multiple regions as long as each region contains a complete copy of the file. This provides for routing accesses to frequently used files. Data Placement is a crucial design decision as naive replication can compromise performance and reduce the efficacy of our approach. The proposed region-aware placement policy takes into account the different regions and distributes the three default replicas across the regions. We can also observe workload patterns and job queue predictions, and use the APIs to move or pre-stage replicas across regions to ensure that the suitable clusters identified using Algorithm 1 have the needed data.

A problem of cross-region replication is that the write time for a data item may increase. We can mitigate it by relaxing the reliability requirement and returning to the application after writing to the first replica only, while other replicas are created asynchronously. However, given that HDFS is write-once read-many file system, even if we wait for all replicas to be written synchronously, the write overhead is amortized quickly by the performance and data locality advantages achieved using our region-aware placement. Thus, we expect the impact on the overall application execution time due to our HDFS enhancement to be negligible.

E. Hardware-Heterogeneity-Aware Scheduling

ϕSched realizes heterogeneity-aware scheduling as follows.

TABLE I
ϕSCHED APIS FOR ENHANCING HDFS WITH REGION INFORMATION.

API	Arguments & Return Type	Description
boolean createFileRegion(...)		**Creates a replica of a file in the specified region.**
	String `filename`	Name of the file to be replicated.
	String `region`	Region in which the replica will be created.
	boolean `return_value`	Returns 0 on success, 1 on failure.
boolean deleteFileRegion(...)		**Removes a replica of a file from a region.**
	String `filename`	Name of the file whose replica will be deleted.
	String `region`	Region from which to remove the replica.
	boolean `return_value`	Returns 0 on success, 1 on failure.
boolean moveRegion(...)		**Moves replicas of a file across regions.**
	String `filename`	Name of the file to be moved.
	String `from_region`	Source region from which replica will be removed.
	String `to_region`	Destination region for the new replica.
	int `number_of_replicas`	Number of file replicas to be moved.
	boolean `return_value`	Returns 0 on success, 1 on failure.
void setRepRegion(...)		**Modifies the replication policy for a file.**
	String `filename`	Name of the file to be affected.
	String `region`	Region for the new replica.
	int `number_of_replicas`	Number of replicas under the new policy.
void findRegion(...)		**Map of block distributions across different regions.**
	String `filename`	Name of the file to be tracked.

Whenever a job is submitted to the job queue, ϕSched invokes the Cluster Manager to compute the expected execution time of the job on the different clusters. The Cluster Manager in turn consults the Execution Predictor to return a sorted list of clusters. ϕSched then uses the list and the enhanced HDFS APIs to find the region that contains the job associated data. This information is then used to select an appropriate cluster to execute the job. The job information is also tracked to guide future analysis and scheduling. In case a cluster, C_a, is available but does not have the required data, ϕSched will wait for a pre-specified time for a cluster with the data to become available. If that does not happen, ϕSched will invoke the HDFS APIs to copy the data to C_a. This ensures that jobs wait time is bounded as long as resources are available in the system.

Profiling execution time for all applications across all the clusters in a deployment will help us understand the behavior of studied applications. Once we have a sorted list of the time that each application takes to complete on a specific hardware, Execution Predictor will be able to estimate the execution time and required resources for upcoming jobs based on this information. This approach enables ϕSched to appropriately schedule each application according to available resources and the performance of the application on a particular hardware.

IV. IMPLEMENTATION

In this section, we describe our implementation of the various components of ϕSched and the HDFS enhancements.

a) Cluster Manager and Execution Predictor: We have implemented a proof-of-concept Cluster Manager and Execution Predictor in Python. We used the *SAR* tool [21] to collect job execution traces containing information such as disk, network, memory, and CPU usage for applications running on various clusters. We also parse the Hadoop logs to determine the timestamps associated with the start and finish time for the applications, which are then used to separate execution information for each application. The Execution Predictor uses the MySQL database to store the collected application information as well as the associated resource utilization. We used MySQLdb module [17] (package name *python-mysqldb*) to enable this interaction.

b) Region Identification: The HDFS region-awareness is realized by modifying or adding about 1800 lines of Java code in Hadoop to add the features of and to enable the APIs of Table I. We introduce a new parameter *dfs.region.id* in the Hadoop configuration file (`hdfs-site.xml`), which the cluster administrator can use to identify the region to which different DataNodes belong. Next, we modify HDFS's *DataNodeDescriptor* data structure to incorporate the unique identifier as an additional global characteristic of each DataNode. The extended descriptor can then be used by the HDFS's *DataNodeRegistration* process for registering the region-based DataNode with the NameNode.

To support region based data placement, we modify the NameNode's *ReplicationTargetChooser* component to implement the proposed region-aware data placement. A list of nodes is chosen from the *NetworkTopology* structure that provides information about various racks and regions in the cluster (*clusterMap*). The nodes selected to store a replica of a data is added to the *excludenode* list to ensure that multiple replicas of a block are not placed on the same node.

After a DataNode is chosen to store a block, the *block* and its corresponding *INodeFile* structure are associated with the DataNode's region. This is to enable re-replication of the block in the same region in case of a failure. A background daemon periodically runs to ensure that the blocks are associated with appropriate regions, and if not, the daemon initiates our *moveRegion* API to move the replicas to the appropriate regions.

V. EVALUATION

We evaluate ϕSched using a real deployment on a medium-scale cluster. In the following, we first study the characteristics of 12 representative Hadoop applications on four different cluster hardware configurations. Next, we evaluate the impact of our HDFS enhancement and data placement policy. Finally,

TABLE II
HARDWARE CONFIGURATIONS CONSIDERED IN OUR EXPERIMENTS. VCPUS: VIRTUAL CPU; ECU: EC2 COMPUTE UNIT; 1 ECU'S EQUIVALENT CPU CAPACITY IS 1.0-1.2 GHZ 2007 OPTERON PROCESSOR.

Name	ECUs	vCPUs	RAM (GB)	Storage (GB)	Network
m3.large	6.5	2	7.5	1 x 32 (SSD)	Moderate
m3.xlarge	13	4	15	2 x 40 (SSD)	High
m2.xlarge	6.5	2	17.1	1 x 420	Moderate
c1.xlarge	20	8	7	4 x 420	High

TABLE III
REPRESENTATIVE MAPREDUCE (HADOOP) APPLICATIONS USED IN OUR STUDY.

Application	Map		Reduce	Number	
	Input	Output	Output	Mapper	Reducer
NutchIndex	1.5 GB	2.8 GB	1 GB	1	81
WordCount	6 GB	30 GB	12 KB	102	8
DFSIOE-Read	8 GB	–	–	128	1
DFSIOE-Write	8 GB	–	–	128	1
Kmeans	1 GB	64 KB	1 GB	20	1
Hive-bench	5 GB	3.2 GB	256 MB	8	16
PageRank	128 MB	1 GB	12.5 MB	16	8
Bayes	128 MB	256 KB	4.5 GB	16	1
RandomWriter	–	–	3 GB	32	0
Sort	3 GB	11.5 GB	3 GB	64	8
TeraGen	–	–	15 GB	16	0
TeraSort	15 GB	15 GB	15 GB	249	8

we compare the overall ϕSched performance against a hardware oblivious workflow scheduler.

A. Experimental Setup

We used the Amazon EC2 [1] to perform our experiments. We used four clusters of eight homogeneous nodes, where each cluster had a different hardware configuration as listed in Table II[3]. All the virtual machines that we use are based on 64-bit Ubuntu Server 12.04.3. In all of the Hadoop deployments considered in our tests, the master node ran both the Hadoop JobTracker and NameNode, and was co-located with a worker node. Moreover, all worker nodes were configured with two map slots and two reduce slots, along with a DataNode component.

B. Studied Applications

In this section, we describe 12 applications from the well-known Hadoop HiBench Benchmark Suite [19], which we have used in our study. These applications are representative of batch processing jobs, iterative jobs and interactive querying jobs. Table III lists the applications, and for each also summarizes parameters such as the input and output data size, and the number of mappers and reducers.

RandomWriter: is a map-only application where each map task takes as input a name of a file and writes random keys and values to the file. There is no intermediate output, and the reduce phase is an identity function.

TeraGen: generates a large number of random numbers, and is typically used for driving sorting benchmarks. This is a also a map-only application and does not take any input.

[3]Amazon EC2 description does not specify the exact networking characteristics, rather provide a relative ranking only, which we report in the table.

WordCount: counts the frequency of all the different words in the input. The map task simply emits $(word, 1)$ for each word in the input, a local-combiner computes the partial frequency of each word, and the reduce tasks perform a global combine to generate the final result.

Sort: performs a sort of the input. A mapper is an identity function and simply directs input records to the correct reducer. The actual sorting happens thanks to the internal shuffle and sort phase of MapReduce, thus the reducer is also an identity function.

TeraSort: samples the input data and estimates the distribution of the input. It performs sampling-based partitioning of data, which also provides for an even distribution of input across reducers so as to achieve a scalable MapReduce-based sort. The sorting is achieved similarly as in *Sort*.

NutchIndex: is representative of a large-scale search indexing system. Nutch is a subsystem of the Apache search engine [26], which crawls the web links and converts the link information into inverted index files.

PageRank: is a key component of a web search workflow. It iteratively calculates representative score for each page, P , by adding the scores of all pages that refer to P. The process is iterated until all scores converge.

Kmeans: takes a set of points in an N-dimensional space as an input, and groups the points into a set number of clusters with approximately an equal number of points in each cluster.

Bayes: is a popular classification algorithm for knowledge discovery and data mining and is a part of Mahout distribution [23]. Bayes implements the training module for the naive Bayesian knowledge discovery algorithm atop Hadoop.

HiveBench: is representative of analytic querying on Hive [31], a parallel analytical database built on top of Hadoop. The benchmark performs join and aggregate queries over structured data.

DFSIOE-Write: is a micro benchmark that uses a specified number of Hadoop tasks to perform parallel writes to HDFS and reports the measured write throughput.

DFSIOE-Read: is a similar to *DFSIOE-Write* except that it performs simultaneous reads to the data generated by *DFSIOE-Write*.

C. Application Analysis

In our first set of experiments, we analyze the performance of our test applications under four different clusters configurations. The input parameters of the application are specified in Table III. The results discussed below are average of four executions; the standard deviation across the runs was observed to be negligible.

1) Performance Comparison: In this test, we measured the execution time of our test applications on the studied clusters. As shown in Figure 3, the execution time of the applications varies across different cluster configurations. We find that across all applications, on average, *m3.xlarge* performs 17.5% faster than both *m3.large* and *m2.xlarge* cluster configurations. However, we observe that the variation in performance is not similar across all applications. For instance, in case of

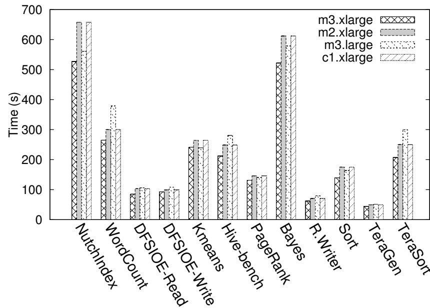


Fig. 3. Application execution time on the studied hardware configurations.

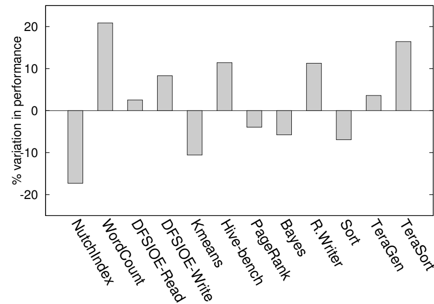


Fig. 4. Performance improvement observed on *m3.large* compared to *m2.xlarge*.

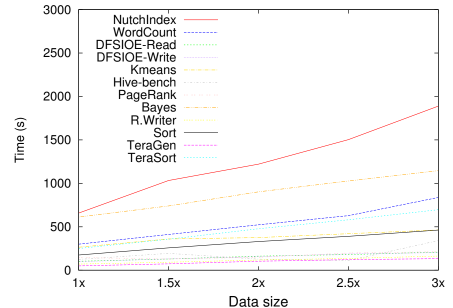


Fig. 5. Effect of increasing the input data size on execution time.

NutchIndex, *m3.xlarge* performs 48% faster than *c1.xlarge*, whereas for the same cluster, *Bayes* perform only 6% faster.

To study this variation in detail, we compared the performance under *m3.large* and *m2.xlarge* across all the studied applications. Figure 4 shows the results. For applications such as *NutchIndex*, *Kmeans*, *PageRank*, *Bayes*, and *Sort*, *m3.large* performs better, while for the rest of the applications *m2.xlarge* performs better. One reason for this is the varying resource needs of the applications. For example, *TeraSort* that is a memory intensive application performs 16.5% faster in *m2.xlarge* that has more memory, and *NutchIndex* that involves significant network and I/O usage performs better in *m3.large* that has better interconnects. Similar pattern is also observed while comparing the execution time of *m3.xlarge* with *c1.xlarge*, in fact the best case placement will perform 34% faster than the worst case. These results validate our claim that the performance of the application varies significantly across various cluster configurations, and can be problematic if the entire deployment is managed using a single Hadoop instance or in a hardware oblivious manner.

2) Impact of Data Size: In the next set of experiments, we study the impact of increase in data size on the performance of the studied applications. Figure 5 shows the execution time of the applications under varying data size for the *m2.xlarge* configuration. We increase the input size shown in Table III from 1× to 3×. We find that although the increase in the execution time is linear, the rate of increase is not the same across the applications, e.g., *PageRank* takes 1.31× the time to process 3× more data, whereas *NutchIndex* takes 2.87× the time. Understanding the scaling factor for an application enables us to better estimate the time and resources required by the application to execute on a particular hardware configuration with a given data set size. We note that a similar performance-data size pattern was also observed under other hardware configurations, though the rate of increase in application execution time varied across the hardware configurations.

3) Usage Characteristics: Next, we study the CPU, memory, storage, and network usage of our test applications. While, we studied all the applications and observed similar variations, we present the results only for *Kmeans*, *TeraSort*, and *Bayes*.

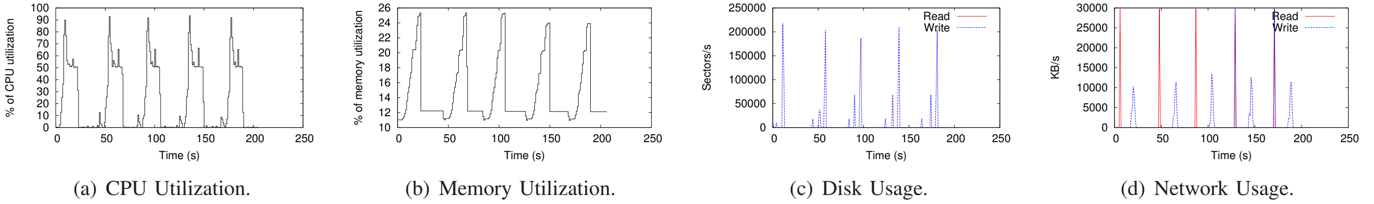
Kmeans is an iterative application and the size of the data does not vary between iterations. Moreover, Figure 6 shows that the resource usage is similar across iterations. We observe that *Kmeans* is CPU bound and uses almost 28% of the CPU, on average. Next, Figure 7 shows the usage characteristics of *TeraSort*. Although the execution time of the application is similar to that of *Kmeans*, it shows 35% increase in the CPU usage and 24% increase in memory utilization with the peak memory utilization reaching up to 50% compared to that for *Kmeans*. Similarly, while comparing the storage and network usage of *TeraSort* and *Kmeans*, we observe that *TeraSort* shows higher usage characteristics with up to 9× for storage and 10× for the network on average. The peak usage reaches up to 2× for storage and 4× for the network. Thus, execution time alone is not a good indicator of the suitability of a resource to efficiently support an application.

The results for *Bayes* are shown in Figure 8. We see that *Bayes* has 2× the execution time compared to both *Kmeans* and *TeraSort*. We observe that in spite of the increased execution time, the average resource utilization is very low. The peak memory usage is less than 20% throughout the execution of the application, and it is 5× less than that of the peak utilization of *TeraSort*. Moreover, the average memory utilization is 6× lower than that of *TeraSort*. Similarly, the average utilization of CPU, storage and network is also low and bursty. Understanding such usage behavior enables us to co-locate appropriate tasks, e.g., a compute-intensive task with an I/O-intensive task running on the same cluster, in order to achieve efficient resource usage without sacrificing application performance.

D. HDFS Enhancement

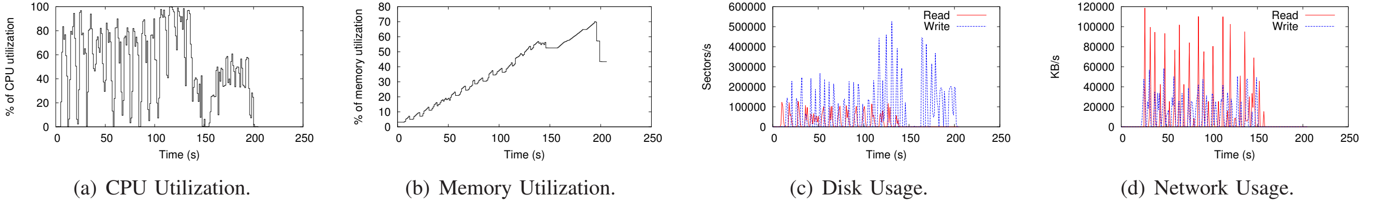
In the next experiment, we evaluate how our HDFS enhancements impact ϕSched. For this test, we use a local cluster instead of EC2. This is because we want to use a slower interconnect that can help highlight the impact on the network, which may be masked due to the high-speed interconnects in EC2. To emulate a large cluster with a large number of DataNodes, we run five DataNodes in each physical node, which gives us a total of 50 DataNodes. We categorize the nodes into five regions.

1) Validation of Placement Policy: To validate our region-aware placement policy, we ran *TeraGen* to generate 20 GB (318 blocks) of data distributed across different nodes. Figure 9 shows the distribution of blocks, which we determine by



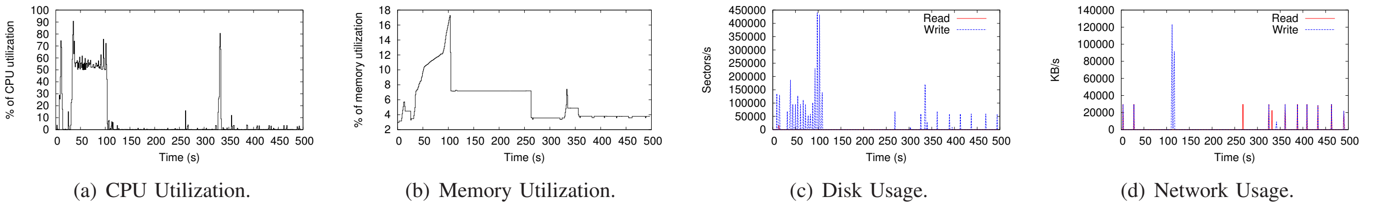
(a) CPU Utilization. (b) Memory Utilization. (c) Disk Usage. (d) Network Usage.

Fig. 6. Resource usage characteristics of *Kmeans* on *m3.xlarge*.



(a) CPU Utilization. (b) Memory Utilization. (c) Disk Usage. (d) Network Usage.

Fig. 7. Resource usage characteristics of *TeraSort* on *m3.xlarge*.



(a) CPU Utilization. (b) Memory Utilization. (c) Disk Usage. (d) Network Usage.

Fig. 8. Resource usage characteristics of *Bayes* on *m3.xlarge*.

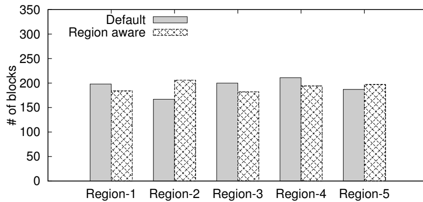


Fig. 9. Distribution of data blocks across different regions under default and region-aware policies.

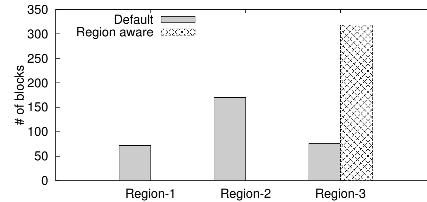


Fig. 10. Number of blocks that are replicated across multiple regions under default and region-aware policies.

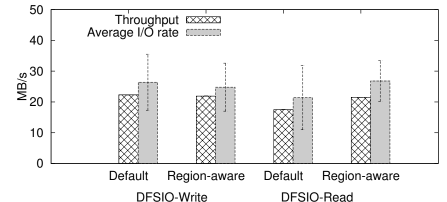


Fig. 11. Overall write throughput and average I/O rate per map task in *TestDFSIOE-Write* and *TestDFSIOE-Read* under default and region-aware policies.

parsing HDFS logs. We find that both the default placement and region-aware placement distributes the data uniformly among different regions. However, a closer analysis (Figure 10) reveals that for the default policy, only 24% of the files have all their blocks replicated across three different regions, and more than 22% of the files have their data blocks replicated within only one region. This would lead to expensive remote accesses if the jobs are scheduled to the clusters associated with the other four regions. In contrast, the region-aware policy distributed all the blocks across different regions, thereby providing a more efficient distribution of data, which in turn would reduce the network overhead when jobs are scheduled across regions.

To ensure this is the case, we took the data placement distributions created by the two policies, and ran *TeraSort* on the distributed data. We observed that the default policy resulted in 48% accesses that are remote reads, whereas region-aware policy has only 14% remote reads. This eliminates the additional network overhead for 34% of reads, consequently improving the overall performance.

2) Performance Analysis: In our next test, we measured the read and write performance under our HDFS enhancements using the HDFS benchmark *TestDFSIO*. Here, each worker node writes a 1024 MB file (16 blocks) during the write test followed by reads of a file of the same size during the read test. Figure 11 shows the overall I/O throughput for each of the map tasks, as well as the average I/O rate across all map tasks. We find that the HDFS enhancements yield lower throughput and average I/O rate for the write operations. This is because of the network overhead involved in writing all the replicas to different racks. As pointed our earlier, this overhead can be amortized due to write-once read-many workloads of Hadoop, as well as through use of asynchronous replication. In case of read operations, the region-aware placement shows 23% improvement in throughput and 26% improvement in average I/O rate. Moreover, we find that the variance in the average I/O

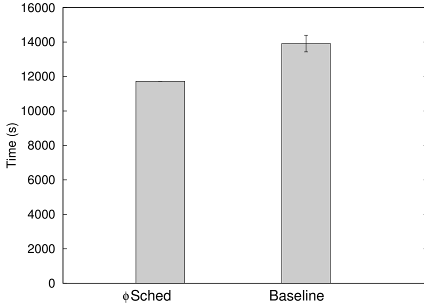


Fig. 12. Execution time of the test workflow under ϕSched and hardware oblivious workflow scheduler.

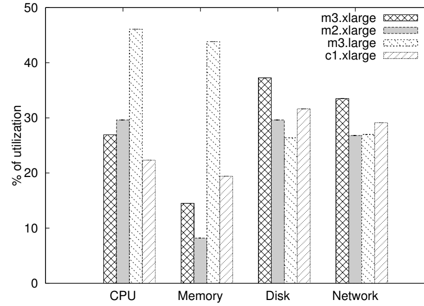


Fig. 13. Average hardware usage of the test workflow under hardware oblivious scheduler.

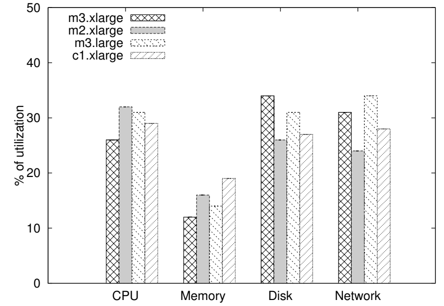


Fig. 14. Average hardware usage of the test workflow under ϕSched.

rate for the default policy is high because of the high variation in the network overhead associated with each read operation. As observed, this is minimized under the region-aware policy.

These experiments outline the benefits that the proposed region-aware HDFS enhancements can provide for Hadoop workflow scheduling.

E. Performance of ϕSched

In our next experiment, we evaluate the performance of ϕSched. We use a 20-node Hadoop deployment with the four cluster configurations of Table II, each with five nodes. The Hadoop master components co-exist with a worker node in each of the clusters. Moreover, we run the management components of ϕSched on a separate *m3.xlarge* node. For comparison, we use a hardware oblivious workflow scheduler (similar to schedulers such as Oozie or Nova) as our baseline, where data is randomly assigned to clusters and workloads are scheduled to clusters that store the input data whenever possible. To drive our test, we generated a large workflow comprising of 48 applications chosen randomly from Table III. Each of the application was included at least once, with multiple instances of the same application processing randomly varying input data. First, we measured the execution time for baseline and ϕSched as shown in Figure 12. We repeated the experiment three times. The baseline scheduler results varied, while those for ϕSched were consistent across the runs. We observe that ϕSched yields 17% to 22% better execution time than that under baseline, and the average improvement is observed to be 18.7%.

Next, we compared the average resource utilization across different hardware configurations under baseline and ϕSched. Figures 13 and 14 show the results. We find that the average memory utilization for the high-memory cluster *m2.xlarge* is lower than that of *m3.large* under baseline. Similarly, *c1.xlarge* that is provisioned only with a HDD performs 1.2$\times$ more I/O operations than *m3.xlarge* that is provisioned with a SSD. This leads to an increased execution time. In contrast, ϕSched considers resource availability while scheduling the jobs to different clusters, which results in better utilization of available resources as seen in the figures. For example, the average memory utilization in *m2.xlarge* is 1.5$\times$ higher than *m3.large*, which is the expected behavior.

In summary, our evaluation of ϕSched reveals that hardware-aware scheduling is a viable solution in large deployments with multiple heterogeneous clusters. ϕSched can improve the execution time of the applications by scheduling jobs to clusters that are better suited to support them. These features are key to sustaining Hadoop for emerging architectures and applications.

VI. Related Work

Several recent works [40], [8], [11] integrate workflow management in Hadoop. Apache Oozie [20] is a popular workflow scheduler for Hadoop jobs that considers availability of job-specific data and the completion of dependent events in its scheduling decisions. Cascading [35] supports a data flow paradigm that transforms the user generated data flow logic into Hadoop jobs. Similarly, Clustera [13] extends Hadoop to handle a wide variety of job types ranging from long running compute intensive jobs to complex SQL queries. Nova [27] workflow manager uses Pig Latin to deal with continually arriving data in large batches using disk-based processing. Kepler+Hadoop [33] is another high-level abstraction built on top of Hadoop, which allow users to compose and execute applications in Kepler scientific workflows. Percolator [28] performs transactional updates to data sets, and uses triggers to cascade updates similar to a workflow. These works are complementary to ϕSched in that they provide means for handling different types of workflows. However, ϕSched is unique in its hardware-aware application scheduling, which to the best of our knowledge has not been attempted by any of the existing works for Hadoop workflows. We note that our HDFS enhancements can co-exist with other Hadoop workflow schedulers as well.

There has been work [15], [4], [38] on hardware-heterogeneity-aware workflow scheduling for High Performance Computing (HPC) workloads. However, these have not been extended to the Hadoop ecosystem, and given the inherent differences in HPC and Hadoop cluster architectures, cannot be simply applied to Hadoop.

HDFS data placement has also been explored [34], [16], [37], [22]. However, such works do not optimize the inter-cluster data movement. Moreover, ϕSched's HDFS extension

APIs are unique and provide efficient means to support workflow scheduling on a heterogeneous computing substrate. Yahoo! acknowledges the existence of "common data sets" across multiple cluster and proposes a framework [30] for multiple Hadoop clusters to share these data sets. However, unlike ϕSched, such data sharing still involves explicit copying of the data across different clusters, which is expensive compared to our approach.

VII. Conclusions

In this paper, we design and implement ϕSched, a novel hardware-aware workflow scheduler for Hadoop. We observe that different workflows perform differently under varying cluster configurations, and making workflow managers aware of the underlying configuration can significantly increase overall performance. To implement this approach, we developed a hierarchical scheduler that treats a Hadoop deployment as a collection of multiple heterogeneous clusters. We also enhance HDFS to manage storage across a multi-cluster deployment, which allows ϕSched to handle data locality as well as enable pre-staging of data to appropriate clusters as needed. We study the impact of ϕSched on Hadoop application performance using a range of representative applications and configuration parameters. Our evaluation shows that ϕSched managing four different clusters can achieve performance improvement of 18.7%, on average, compared to hardware oblivious scheduling. Moreover, for the well-known *TestDFSIO* benchmark, our HDFS enhancement increased the I/O throughput by up to 23% and the average I/O rate by up to 26%.

In our future work, we aim to extend ϕSched to handle clusters comprising heterogeneous architectures as well, such as those comprising ARM-based microservers, GPUs and accelerators.

Acknowledgment

This work is sponsored in part by the NSF under the CNS-1016793 and CCF-0746832 grants.

References

[1] E. Amazon. Amazon elastic compute cloud (amazon ec2). 2010.
[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proc. USENIX NSDI*, 2012.
[3] Apache Software Foundation. Hadoop, 2011. http://hadoop.apache.org/core/.
[4] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *Proc. IEEE CCGrid*, 2005.
[5] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.
[6] D. Borthakur. Facebook has the world's largest hadoop cluster!, 2010. http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html.
[7] D. e. Borthakur. Apache hadoop goes realtime at Facebook. Proc. ACM SIGMOD, 2011.
[8] Q. Chen, L. Wang, and Z. Shang. Mrgis: A mapreduce-enabled high performance workflow system for gis. In *Proc. IEEE International Conference on eScience*, 2008.
[9] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endowment*, 5(12):1802–1813, 2012.
[10] E. S. Chung, J. D. Davis, and J. Lee. Linqits: Big data on little clients. In *Proc. ACM ISCA*, 2013.
[11] F. Corona. Facebook, 2012. https://gigaom.com/2012/11/08/facebook-open-sources-corona-a-better-way-to-do-webscale-hadoop.
[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[13] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *Proc. VLDB Endowment*, 1(1):28–41, 2008.
[14] F. Dong. *Extending Starfish to Support the Growing Hadoop Ecosystem*. PhD thesis, Duke University, 2012.
[15] F. Dong and S. G. Akl. Pfas: a resource-performance-fluctuation-aware workflow scheduling algorithm for grid computing. In *Proc. IEEE IPDPS*, 2007.
[16] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. *Proc. VLDB Endowment*, 4(9):575–585, 2011.
[17] C. Esterbrook. Using mix-ins with python. *Linux Journal*, 2001(84es):7, 2001.
[18] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proc. CIDR*, 2011.
[19] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, 2010.
[20] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proc. ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 2012.
[21] Linux man page. sar(1), 2013. http://linux.die.net/man/1/sar.
[22] N. Maheshwari, R. Nanduri, and V. Varma. Dynamic energy efficient data placement and cluster reconfiguration algorithm for mapreduce framework. *Future Generation Computer Systems*, 28(1):119–127, 2012.
[23] A. Mahout. Mahout, 2010.
[24] R. Mantri, R. Ingle, and P. Patil. Scdp: Scalable, cost-effective, distributed and parallel computing model for academics. In *Proc. ICECT*, 2011.
[25] M. Mihailescu, G. Soundararajan, and C. Amza. Mixapart: Decoupled analytics for shared storage systems. In *Proc. USENIX HotStorage*, 2012.
[26] A. Nutch. Nutch, 2010. http://nutch.apache.org.
[27] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, et al. Nova: continuous pig/hadoop workflows. In *Proc. ACM SIGMOD*, 2011.
[28] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. USENIX OSDI*, 2010.
[29] S. Radia and S. Srinivas. Scaling hdfs cluster using namenode federation. *HDFS-1052, August*, 2010.
[30] S. Rao, B. Reed, and A. Silberstein. Hotrod: Managing grid storage with on-demand replication. In *Proc. IEEE ICDEW*, 2013.
[31] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endowment*, 2(2):1626–1629, 2009.
[32] C. Wang, X. Li, X. Zhou, Y. Chen, and R. C. Cheung. Big data genome sequencing on zynq based clusters. In *Proc. ACM SIGDA*, 2014.
[33] J. Wang, D. Crawl, and I. Altintas. Kepler + hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. Proc. WORKS, 2009.
[34] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng. Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster. In *Proc. IEEE CLUSTER*, 2010.
[35] C. Wensel. Cascading: Defining and executing complex and fault tolerant data processing workflows on a hadoop cluster, 2008.
[36] T. White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.
[37] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Proc. IEEE IPDPSW*, 2010.
[38] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.
[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. USENIX OSDI*, 2008.
[40] C. Zhang and H. De Sterck. Cloudwf: A computational workflow system for clouds based on hadoop. In *Cloud Computing*, pages 393–404. Springer, 2009.