Exploiting Heterogeneity in Distributed Software Frameworks

Krishnaraj K. Ravindranathan

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

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Chair
Chris Gniady
Narendran Ramakrishnan
Eli Tilevich
Chao Wang
Danfeng Yao

October 28, 2015
Blacksburg, Virginia, USA

Keywords: Distributed Software Framework, MapReduce, Workflow Manager, Heterogeneous Resource Management, Storage Management.

Exploiting Heterogeneity in Distributed Software Frameworks

Krishnaraj K. Ravindranathan

ABSTRACT

The objective of this thesis is to address the challenges faced in sustaining efficient, high-performance and scalable Distributed Software Frameworks (DSFs), such as MapReduce, Hadoop, Dryad, and Pregel, for supporting data-intensive scientific and enterprise applications on emerging heterogeneous compute, storage and network infrastructure. Large DSF deployments in the cloud continue to grow both in size and number, given DSFs are cost-effective and easy to deploy. DSFs are becoming heterogeneous with the use of advanced hardware technologies and due to regular upgrades to the system. For instance, low-cost, power-efficient clusters that employ traditional servers along with specialized resources such as FPGAs, GPUs, powerPC, MIPS and ARM based embedded devices, and high-end server-on-chip solutions will drive future DSFs infrastructure. Similarly, high-throughput DSF storage is trending towards hybrid and tiered approaches that use large in-memory buffers, SSDs, etc., in addition to disks. However, the schedulers and resource managers of these DSFs assume the underlying hardware to be similar or homogeneous. Another problem faced in evolving applications is that they are typically complex workflows comprising of different kernels. The kernels can be diverse, e.g., compute-intensive processing followed by data-intensive visualization and each kernel will have a different affinity towards different hardware. Because of the inability of the DSFs to understand heterogeneity of the underlying hardware architecture and applications, existing resource managers cannot ensure appropriate resource-application match for better performance and resource usage.

In this dissertation, we design, implement, and evaluate Derby*hat*S, an application-characteristics-aware resource manager for DSFs, which predicts the performance of the application under different hardware configurations and dynamically manage compute and storage resources as per the application needs. We adopt a quantitative approach where we first study the detailed behavior of various Hadoop applications running on different hardware configurations and propose application-attuned dynamic system management in order to improve the resource-application match. We re-design the Hadoop Distributed File

System (HDFS) into a multi-tiered storage system that seamlessly integrates heterogeneous storage technologies into the HDFS. We also propose data placement and retrieval policies to improve the utilization of the storage devices based on their characteristics such as I/O throughput and capacity. Derby*hat*S workflow scheduler is an application-attuned workflow scheduler and is constituted by two components. ϕSched coupled with ϵSched manages the compute heterogeneity and DUX coupled with AptStore manages the storage substrate to exploit heterogeneity. Derby*hat*S will help realize the full potential of the emerging infrastructure for DSFs, e.g., cloud data centers, by offering many advantages over the state of the art by ensuring application-attuned, dynamic heterogeneous resource management.

Dedication

Dedicated to my family,
for their endless love, encouragement and support ...

Acknowledgments

I owe my gratitude to a great many people who have contributed toward making this thesis possible and because of whom my graduate experience has been one I will forever cherish.

I am deeply indebted to my advisor Dr.Ali R Butt for his trust, guidance and friendship during my tenure as a graduate student at Virginia Tech. He has always found time in his busy schedule for countless discussions and feedback sessions and has been a strong pillar of support for me. He has never clipped my wings and has always given me the freedom to fly high and explore new ideas. I am forever grateful to him for his assistance in fulfilling my goals that are very dear to me. Beyond being just my advisor, there have been times when he has also donned the hat of a compassionate mentor, helping me up through some tough times both in my personal and professional life. His counsel has been instrumental in helping me recover whenever my steps have faltered. I would like to also personally thank the other members in my dissertation committee: Dr. Chris Gniady, Dr. Naren Ramakrishnan, Dr. Eli Tilevich, Dr. Chao Wang and Dr. Danfeng Yao, for their valuable comments and suggestions. It is indeed a great honor for me to have them serving in my committee.

I am especially grateful to Dr. Tilevich for his guidance on a wide variety of topics ranging from health to job interviews. Our discussions have always been insightful and thought-provoking ones and have added a different perspective to my PhD journey and to my ideas on life as a whole. I am also grateful to Dr. Wang for sharing his knowledge and enlightening me while working together on Pythia. The discussions I had with him evolved into papers that eventually became an integral part of my thesis. I would like to further extend my thanks to all the educational institutes (St.John's, Salem; GGMHS, Salem; Kongu Engineering College, Perundurai; SUNY, Albany; and Virginia Tech, Blacksburg) and teachers who have played path-defining roles during different phases of my academic life.

Through Distributed Systems and Storage Lab (DSSL) at VT, Ali has created the perfect ecosystem for self-development filled with camaraderie and selflessness, much of which transcends from the leader. The DSSL experience is invaluable and will always remain close to

my heart. I enjoyed my time at DSSL to the fullest and will always cherish the moments and be proud of it. Many thanks to all my peers Mustafa, Henry, Guanying, Min, Hafeez, Alex, Kevin, Hyogi, Puranjoy, Nikhil, Sushil, Ali, Safdar, Luna, Uday, Jin and Bharti. It has been an honor to work with each and every one of you. I would like to thank the staff of the VT Computer Science department for their support , particularly the techstaff Rob and Ryan and the administrative staff Debbie, Sharon, Megan, Emily and Melanie. I would also like to extend my thanks to janitors Kathy and David, for maintaining our lab clean and spotless.

I am thankful to all my friends at VT for their support and belief in me, especially, Jai, Harshi, Ananth, Subbu, Kiran, Gayathri, Neelima, Nabeel, Vignesh, Balaji, Shouvik, Sriram, Rubashri, Nabanitha, Karthik and Phillip. I would also like to thank my well-wishers, Shanmu, Kothai, Ashwin, Muthu, Kushal, Arun, Kalish, Ezhil, Bharthi, Kanika, Sindhu and Krishna Prabha, without whom this journey would not have been the fun one that it was. Finally, and most importantly, the love and patience of my family has been and will continue to remain the backbone of all my endeavors. My parents, Ravindranathan and Sivakami, my siblings, Uma and Geetha have showered their unconditional love and encouragement all throughout my life. Their support has been invaluable during my academic life, particularly in my difficult middle and high school days. I would be failing in my duty if I did not mention Visu, Gopi, Sanjeev, Guhan, Gunan and Sahana who have brought more meaning to my life and have helped me appreciate it to the fullest. I would like to express my heartfelt gratitude to my extended family as well and a special mention for Dr.Shankar, Dr.Shoba and Kumar for making me feel at home during the first one and a half years of my time in the US. I greatly value the discussions we had during this time which went a long way in motivating and preparing me for a PhD degree.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DAG Directed Acyclic Graph

DAS Direct Attached Storage

HDD Hard Disk Drives

HDFS Hadoop Distributed File System

HPC High Performance Computing

LSN Localized Storage Node

MTBF Mean Time Between Failures

NAS Network Attached Storage

PPA Popularity Prediction Algorithm

SLA Service Level Agreement

SLO Service Level Objectives

SOC Server on Chip

SSD Solid State Disks

UDF User Defined Functions

URI Uniform resource identifiers

USS Unified Storage System

Chapter 1

Introduction

Large DSF deployments in the cloud continue to grow in both size and number [1–8], given the DSFs are cost-effective and easy to deploy. In recent years, a broad range of disciplines that regularly employ high-performance including Genomics, Astronomy, Business Informatics, High-Speed Physics, and Meteorology are faced with complex data-intensive problems that need to be solved at scale. DSFs such as MapReduce [9]/Hadoop [10, 11] have proved to be promising, regularly supporting such data-intensive applications both in industry [9,12–22] and academia [23,24]. The MapReduce paradigm also serves as a building block for a plethora of more complex DSFs such as Pig [25], HBase [26], Hive [27], Pregel [28], and Cassandra [29]. DSFs such as MapReduce [30] and Hadoop [10] have become the de-facto framework for large-scale data processing and analytics. This is mainly due to the ability of the framework to efficiently handle both large batch processing workloads, such as building search indexes, and short interactive jobs, such as ad hoc data mining queries. The key component enabling the scalability and performance of big data applications is the underlying Hadoop Distributed File System (HDFS), which offers desirable scale-out ability without performance degradation and while ensuring data availability and fault tolerance. A major challenge faced by researchers and IT practitioners in sustaining Hadoop clusters is evolving the storage and I/O infrastructure to deal with the exponentially growing data volumes, and to do so in an economically viable fashion. This is non-trivial, especially as the network bandwidth provided by the cluster networking infrastructure is growing by an order of magnitude faster than the I/O bandwidths of hard disk drives (HDDs) [31] . In a typical large-scale Hadoop deployment, the intra-rack and inter-rack networks have a bandwidth 200× and 400× that of the disk bandwidth [31], respectively.

A promising trend in storage technologies is the emergence of heterogeneous and hybrid storage systems [32–35] that employ different types of storage devices, e.g., SSDs, ramdisks, etc. Moreover, the networking infrastructure bandwidth is growing at a pace that is an order

of magnitude higher than the I/O bandwidth improvements in hard disk drives (HDDs) [36]. The impact of SSDs on the performance of MapReduce applications has recently captured the research community's attention. These two trends are enabling the realization of distributed, hierarchical, hybrid and heterogeneous storage solutions that are efficient and cost effective, e.g., Hystor [34] and ConquestFS [37]. These systems typically integrate HDDs with fast emerging storage mediums, e.g., ramdisks, SSDs, etc. Faster storage serves as a buffer for frequently accessed data and yields very high I/O rates, while the HDDs store the infrequently accessed data and provide cheap high-capacity storage for the large data volumes. Recent research [38–41] has shown that SSDs are a viable alternative to HDDs for DSF I/O. However, simply replacing HDDs with SSDs is not practical. Moreover, the hot data is too large to fit in RAM and the cold data is too large to easily fit entirely in flash memory [40]. Thus, adding a flash tier can improve overall performance. This approach is promising, but introduces the challenge of effectively managing the distribution of data among different tiers and selecting a tier for servicing read/write requests with the goal of improving application efficiency.

Recent analysis of Hadoop workloads shows that there is significant heterogeneity in I/O access patterns. GreenHDFS [42] observed a news server like access pattern in job history logs from Bing, where recent data is accessed more than old data and more than 60% of used capacity remained untouched for at least one month (period of the analysis). Scarlett [43] analyzed HDFS audit logs from Yahoo! production clusters and observed that 12% of the most popular files are accessed over ten times more than the bottom third of the data, while more than 1.5% of all files have at least 3 concurrent accesses. These trends highlight that the access pattern based storage policy can not only improve the storage efficiency but also performance.

Another major obstacle to sustaining DSF at scale is that the energy footprint of the large clusters and data centers that support DSF is increasing sharply [42] and imposes significant financial burden [44]. To mitigate this, microserver-based clusters have been proposed as an alternative [45–47]. Microservers are networked embedded devices designed specifically for low powered environments. They employ the Server-on-Chip (SoC) approach to have a CPU, networking, and I/O fully integrated onto a single server chip [45]. Microservers can be synthesized around numerous embedded architectures such as PowerPc [48], ColdFire [49], Mips [50] and ARM [51]. These devices are cheaper compared to traditional servers, energy efficient, and are becoming readily available.

Moreover modern data analysis application workflows are becoming more intricate, and now

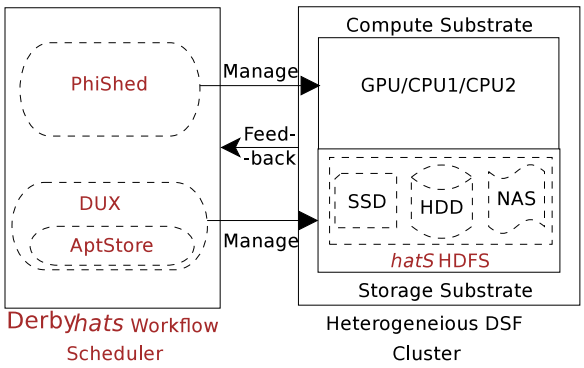


Figure 1.1 Derby*hat*S architecture overview – research contribution are indicated in red color.

comprise complex workflows with a large number of iterative jobs, interactive querying, as well as traditional batch-friendly long running tasks [52]. These multiple independent tasks exhibit different characteristics during application execution, and this coupled with the heterogeneity of underlying infrastructure poses several interesting challenges. DSF computing substrates such as MapReduce have been designed to run in homogeneous environments for applications that are typically composed of a single kernel. Thus, existing feature implementations — such as MapReduce slots and data replica placement — are not capable of exploiting heterogeneity in both the system architecture (different CPUs, embedded devices, GPUs, tiered-storage, etc.) and various stages of a workflow. The proposed research will be able to adapt to such varying application and infrastructure characteristics at runtime to better drive resource management, consequently achieving high performance and efficiency.

1.1 Challenges of Heterogeneous Resource Management in DSFs

A major problem faced in evolving DSFs is to efficiently handle increasing heterogeneity in the underlying infrastructure. For instance, low-cost, power-efficient clusters that employ traditional servers along with specialized resources - such as FPGAs [53, 54], GPU s [55–57], powerPC [58], MIPS [59] and ARM [60, 61] based embedded devices, and high-end server-on-chip solutions [62] - will drive future DSFs infrastructure [63–66]. Similarly, high-throughput DSF storage is trending towards hybrid and tiered approaches [67–71] that use large in-memory buffers, SSDs, etc., in addition to disks. As highlighted in Figure 1.1, the infrastructure that supports DSFs is becoming increasingly heterogeneous. One reason for this is that different types of hardware such as CPUs, memory, storage, and network are

deployed when large clusters typically go through upgrade phases. However, a more crucial driver for the heterogeneity is the rise of hybrid systems. The compute nodes in modern large-scale distributed systems often boasts of multi-core processors with tightly coupled accelerators [72,73]. Numerous current products from major vendors package a few general-purpose cores (e.g., x86, PowerPC) and several accelerators (e.g., SIMD processors, GPUs, FPGAs), yielding power-efficient and low-cost compute nodes with performance exceeding 100 Gflops per chip [74–80]. Therefore, specialized embedded devices are also gaining popularity in DSFs [63–66,81,82]. The net effect of the above trends is that DSF cluster nodes exhibit orders of magnitude variation in terms of compute power, cluster integration and programmability. While recent studies [53,54] have shown that use of specialized accelerators for Hadoop is desirable, sustaining DSFs on such resources is challenging. This is because most modern DSFs are designed to run on homogeneous clusters and cannot effectively handle general purpose workloads [54,83] on heterogeneous resources. For instance, current Hadoop task slots and straggler detection does not support different core types, e.g., one type of CPU vs a faster one or a GPU. Moreover, simply adding more slots to nodes with more compute power is not enough, as these nodes would be starved for data distributed uniformly across all nodes. Hence, powerful nodes would significantly increase network traffic as they run more map tasks on remote data (by stealing from less powerful nodes). Simply skewing the data so more of it will be on powerful nodes would compromise reliability, because the failure of a powerful node would result in the loss of a disproportionately large chunk of the data. While a few recent DSF designs [55, 57] have incorporated GPUs, these still focus only on a single type of accelerator and are thus insufficient for handling the above infrastructure trends.

Similarly, storage systems are increasingly employing hybrid and heterogeneous storage devices such as Solid State Disks (SSD) , Ram Disks and Network Attached Storage (NAS) to yield very high I/O rates at acceptable costs. However DSF storage, such as Hadoop Distributed File System (HDFS) [84], is not equipped to handle such emerging storage systems. This is because all underlying devices are assumed to be comprised of homogeneous storage blocks irrespective of the different I/O characteristics of the devices. For example, an I/O request to HDFS may require access to multiple blocks spanning different storage devices, and the assumption of homogeneity may unpredictably affect performance. The current Hadoop implementation assumes that computing nodes in a cluster are homogeneous in nature and does not account for these specialized architectures. Hadoop scheduling policy assumes homogeneous clusters and enables scheduling decisions on the assumption that all nodes in the cluster have the same processing power and capability, so the computation will

be done at the same rate across all nodes. If a node is slower than the others, it is treated as a faulty machine. Given the large size of clusters dedicated for data-intensive applications, it is not always possible or even desirable to have a large cluster consisting of a single type of machine.

Another problem posed by modern applications is that they typically are complex workflows comprising multiple different kernels [23, 85]. Data analytics workload kernels can be diverse and have heterogeneous resource demands, e.g., some workloads may be CPU-intensive whereas others are I/O-intensive. Some of them might be able to use special hardware like GPUs to achieve dramatic performance gains. Hadoop applications are also becoming more intricate with a large number of iterative jobs, interactive querying, as well as traditional. batch-friendly, long running tasks [52]. Hadoop assumes homogeneity in workloads. Hadoop does not account for the difference of workload characteristics between jobs and thus does not consider the application-resource match while scheduling a job. Hadoop workflows are realized through a variety of high-level tools and languages [86] instead of manual MapReduce programming. Therefore, systems such as Oozie [87], Nova [88], and Hadoop+Kepler [89] 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 interoperability [87]. The extant workflow schedulers are (mostly) oblivious of the underlying hardware architecture. Thus, in their scheduling decisions, the schedulers do not consider varying execution characteristics such as CPU, memory, storage, and network usage of Hadoop applications on heterogeneous computing substrates that are quickly becoming the norm. The complexity and growing data requirements of emerging applications, combined with increasing system heterogeneity and use of specialized resources such as GPU accelerators [55–57] demand innovation to ensure that next-generation DSFs are efficient and can sustain these applications. In the following, we highlight research contributions that we make in this dissertation.

1.2 Research Contributions

The objective of this research is to design, implement, and evaluate an application-characteristics-aware resource manager for DSFs, which adopt a quantitative approach where we first study detailed behavior of various Hadoop applications running on different hardware configurations and propose application-attuned dynamic system management in order to im-

prove the resourceapplication match. Figure 1.1 shows the overall architecture of Derby*hat*S. hat*S* is a novel redesign of HDFS into a multi-tiered storage system that seamlessly integrates heterogeneous storagetechnologies into the HDFS. hat*S* also proposes data placement and retrieval policies, which improve the utilization of the storage devices based on their characteristics such as I/O throughput and capacity. Derby*hat*S workflow scheduler is an application-attuned workflow scheduler and is constituted by two components. ϕSched coupled with ϵSched manages the compute heterogeneity and DUX coupled with AptStore manages the storage substrate to exploit heterogeneity.

1.2.1 LSN

To address the challenges of exploring heterogeneous storage devices such as SAN/NAS in Hadoop, we propose consolidating the disks of each sub-racks compute nodes into a shared Localized Storage Node (LSN) for servicing the sub-rack. An LSN is easier to manage and provision, provides higher I/O performance by having more disks that can be accessed in parallel, and can also be more economical as it employs fewer disks overall than the total used by the sub-racks nodes. We posit that aggregating and sharing resources across a subset of nodes can produce an efficient resource allocation in an otherwise shared-nothing Hadoop cluster . To address this, we present a novel enhancement for Hadoop, which divides a traditional Hadoop rack into several sub-racks, and consolidates the disks attached to each of the sub-racks compute nodes into a shared LSN for servicing the sub-rack. The scope of a single LSN can range from serving a few nodes to perhaps a complete Hadoop cluster rack depending on workload and usage characteristics. An observation that makes the proposed approach viable is that in typical Hadoop clusters, accesses to disks are often staggered in time because of the mix of different types of jobs and data skew across nodes. This, coupled with the bursty node workload, implies that contention at the LSN from its associated nodes is expected to be low. Therefore, by simply re-purposing a sub-racks node-local disks in an LSN, each node can receive a higher instantaneous I/O throughput provided by the larger number of disks in the LSN. Conversely, the LSN can service its associated nodes at the default I/O throughput (experienced by nodes when using their local disks) with fewer numbers of disks at the LSN. We do not argue for provisioning LSNs in addition to the node-local disks, rather placing some or all of the disks from a sub-racks nodes at their LSN. Of course, moving the disks away from a node and into a shared LSN results in loss of data locality, so achieving higher I/O throughput depends on appropriate provisioning of both

disks and network bandwidth to the LSN.

1.2.2 hat*S*

To address the challenges of enabling Hadoop to be aware of the characteristics of the underlying storage, we explore the utility of heterogeneous storage devices in Hadoop and address challenges therein by designing hat*S*, a heterogeneity-aware tiered storage for Hadoop. hat*S* logically groups all storage devices of the same type across the nodes into an associated "tier." A deployment has as many tiers as the different types of storage devices used, and a node with multiple types of devices is part of multiple tiers. For instance, if a deployment consists of nodes with a SSD and a HDD, all the SSDs across the nodes will become part of the SSD tier, and similarly all the HDDs will form the HDD tier. By managing the tiers individually, hat*S* is able to capture the heterogeneity and exploit it to achieve high I/O performance. Contrary to HDFS that only considers network-aware data placement and retrieval policies, hat*S* proposes additional policies to replicate data across tiers in a heterogeneity-aware fashion. This enhances the utilization of the high-performance storage devices by efficiently forwarding a greater number of I/O requests to the faster tier, thus improving overall I/O performances. To facilitate this, in addition to the standard HDFS APIs, hat*S* also provides custom APIs for seamless data transfer across the tiers and management of stored data in each tier. These features allow hat*S* to integrate heterogeneous storage devices into Hadoop to extract high I/O performance.

1.2.3 AptStore

To address the challenges of workload based adaptive storage, we design a tiered storage system, AptStore, with two tiers designed to better match the heterogeneous Hadoop I/O access patterns. We propose using two classes of storage: Primary storage — Direct Attached Storage in Hadoop node for files that require high throughput, and Secondary Storage — NAS for unpopular files and files with lower Service Level Objectives (SLO). AptStore analyzes the I/O access patterns and suggests policies to increase the overall read throughput and the storage efficiency of the system. Our system optimizes for read throughput as typically MapReduce workloads exhibit write-once read-many characteristics [90]. To achieve this, we predict the popularity of each file, and then retain the popular files in primary storage and

move unpopular files to secondary storage. We also adjust the replication factor of files in primary storage based on their popularity to yield higher read throughput. The replication factor for files in the secondary storage is set to 1. However, other means such as RAID are employed in secondary storage to achieve fault tolerance. We have realized AptStore as an extension to the Unified Storage System (USS) [91, 92], a federated file system for Hadoop, which allows transparent movement and management of data across different file systems.

1.2.4 DUX

To address the challenges of providing an application-attuned storage management for Hadoop, we design DUX that employs two tiers designed to better match the heterogeneous Hadoop I/O access patterns. The tiers include a fast SSD tier that aggregates the SSDs provisioned in each node, and a secondary HDD tier comprising of HDDs. We use the SSD tier as a cache in front of the HDD tier. The main insight behind DUX is that performance gains from using SSDs for either HDFS data or intermediate data are dependent on an applications I/O access patterns. The key contribution of DUX is that it profiles application behavior on different storage configurations and proposes an appropriate storage configuration for scheduling the application in the future. DUX provides a holistic solution for effectively orchestrating the SSD tier by performing three major functions. (i) employ AptStore to observe the HDFS I/O access patterns and load popular files into the SSD tier. (ii) Predict the impact of the I/O accesses on execution time and choose an appropriate tier for storing the intermediate data. (iii) For the jobs waiting in the job queue, prefetch the input data into the SSD tier if it has not been selected by AptStore.

1.2.5 ϕSched

To address the challenges of improving the application-resource match, 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. 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, which: i) profiles applications execution on different clusters and performs a 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 [52, 93] 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 and suitability of of the clusters to support the job based on prior profiling. Based on these factors, ϕSched then suggests the best cluster to execute the job.

1.2.6 ϵSched

To address the challenges of improving the energy efficiency of DSFs, we propose to consider applications behavior on specific hardware configurations when scheduling Hadoop workflows. In ϵSched, we propose to improve the energy efficient scheduler by considering heterogeneous Hadoop deployments that comprise of one or more homogeneous sub-clusters. The set of tasks to be executed on the heterogeneous deployment cluster will be scheduled to the sub-clusters such that the total energy consumption is minimized, while the performance goals specified in the Service Level Agreement (SLA) are met. We propose simple, application characteristic-aware task scheduling in Hadoop to reduce the power consumption or to improve the throughput. We present ϵSched, a heterogeneity-aware and power-conserving task scheduler for Hadoop. ϵSched extends our own ϕSched system [94] – a hardware characteristic-aware scheduler that improves the resource-application match. We extend Hadoop's hardware-aware scheduler, which is optimized only for performance, to be an energy efficient scheduler. We adopt a quantitative approach where we first study detailed behavior of applications, such as performance and power characteristics, of various representative Hadoop applications running on four different hardware configurations. Next, we incorporate the findings of these experiments into ϕSched.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. In chapter 2, we discuss the related work and background technologies that lay the foundation of the research conducted in this dissertation. In chapter 3, we study the impact of different storage configurations on Hadoop application performance using a range of representative applications and configuration parameters. In chapter 4, we design and implement a novel heterogeneity-aware and tier-based enhancement for HDFS, hat*S*. This solution also supports tier-aware data storage and retrieval policies to exploit the individual advantages of various types of storage elements. In Chapter 5, we present AptStore, a dynamic data management scheme for Hadoop for achieving higher throughput and lower storage cost. We observe that managing all Hadoop data in a uniform manner results in increased storage overhead or reduced read throughput. In Chapter 6, we present the design and implementation of DUX, an application-attuned dynamic data management system for Hadoop. DUX aims to improve overall I/O throughput of Hadoop via effective use of SSDs as a cache, not for all data, but only for workloads that are expected to benefit from SSDs. In Chapter 7, we develop a hierarchical scheduler, ϕSched, 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. In Chapter 8, we design and implement ϵSched, a novel hardware-aware workflow scheduler for Hadoop that has different performance and power usage characteristics under varying cluster configurations, and make workflow managers aware of the underlying configuration to improve the performance and power consumption. We then conclude the dissertation in Chapter 9 including a discussion of future directions based on our DSF resource management framework.

Chapter 2

Literature Review

As previously described, this research focuses on heterogeneity in compute and storage substrate in Hadoop. This section summarizes the prior work in heterogeneity in Hadoop, as well as state-of-the-art work on improving the storage efficiency in Hadoop.

2.1 MapReduce Model

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 $12 - 24$ [95]) are grouped together and connected with a network switch to form a rack. One or more racks form the Hadoop cluster. Intra-rack network bandwidth in large deployments is typically 20 GB and the inter-rack is 40 GB [31].

The compute component is managed by the *JobTracker* component that accepts jobs from the users and also manages the compute nodes that each run a *TaskTracker*. Each job consists of several map and reduce functions specified by the programmer. 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. All data in MapReduce is represented as key-value pairs [96]. 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 determine the execution time of the application [97]. Although speculative execution [98] can reduce this dependency, it leads to significant resource wastage due to re-execution of tasks.

The data management for a Hadoop cluster is provided by the Hadoop Distributed File System (HDFS). HDFS manages the persistent data associated with the Hadoop cluster such as the input and the output of applications. 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 cluster node. 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.

Hadoop programs typically consist of multiple stages of computation, and involve communication from each stage to the next. Hadoop generates a large amount of temporary data that is not stored in HDFS and is commonly referred to as intermediate data. This data is produced as output from one stage and used as an input for the next stage. For instance, the output of the map stage serves as input to the next reduce stage. This data is write once, read only, and is short-lived and used immediately [99]. This intermediate data is stored locally on the nodes that generate it, and is partitioned and organized into a number of chunks in the shuffle phase. This shuffle data is used as the input to the reduce phase.

Hadoop I/O can benefit from a node-level cache, particularly for HDFS I/O. Reading the same block multiple times can benefit from the DataNodes read cache. However, in a production cluster, the possibility of benefiting from such caching is much smaller and it is difficult to guarantee the availability of all the blocks of a needed file in the cache. With a lot of blocks being read and written, the cache would not be sufficient and essentially generate no hits.

2.2 Hadoop Storage Systems

In this section we provide an overview of prior work in Hadoop with respect to improving the of I/O. We focus on extendable frameworks as well as other state of the work on storage efficiency and performance of HDFS.

2.2.1 Storage Consolidation in Hadoop

Porter [100] proposes consolidating Hadoop node disks into a SuperDataNode and running DataNode instances in virtual machines on the SuperDataNode. While we share the concept of disk consolidation, our work is novel in its observation that resources are not well-utilized in a bursty workload environment due to Hadoop's shared-nothing architecture. Moreover, we examine in detail how consolidating disks can improve the disk utilization, as well as provide means for improving I/O throughput by increasing LSN disk and interconnect provisioning. TritonSort [101] presents a system for sorting vertically, which optimizes the hardware configuration to the sorting algorithm. The main idea is to keep components of the system as balanced as possible, i.e., the utilization of all resources in the system should be maximized, and any resources that slow down the system are taken out. This work is complementary to ours, as we share the idea of optimized utilization. Our focus, though, is on improving the disk utilization of Hadoop applications by consolidating disks into LSNs.

Ganesh et al. [102] argue that neither disk locality nor remote memory access is good enough in data center computing. The authors propose mechanisms that support memory locality, so applications achieve high throughput by mostly employing in-memory accesses. Achieving memory locality is complementary to our work. Although memory locality can make one single task achieve high performance, not all tasks can achieve memory locality at the same time, and we believe disk accesses will still be needed to service a dominant portion of data requests. By consolidating disks into LSNs, we can either support the same throughput with fewer disks, or support higher throughput with the same number of disks. This allows the cluster designers to select a configuration most suited to their application needs.

Many systems such as Swift [103], Zebra [104], GPFS [105], Panasas [106], AptStore [107], and Flat Datacenter Storage [108] try to improve read and write throughput by striping logs, files, and blocks across file servers. Flat Datacenter Storage [108], a filesystem built on top of an advanced network topology, and Camdoop [109], that uses a direct-connect network topology, argue the need for better provisioning and utilization of the disk bandwidth in the modern Big Data applications. Flast Datacenter Storage utilizes disk-to-disk communication and targets to make recovery from disk failures extremely fast. On the other hand Camdoop advocates the need to decrease the traffic instead of increasing the bandwidth. MixApart [110] is another work to reduce the cost and inefficiencies of shared storage systems. Contrary to our work, MixApart proposes a scalable data processing framework in which a single consolidated storage back-end manages enterprise data and services all types of workloads. Mixapart utilizes exchange of scheduling and location information about task

and data within two schdulers designed to handle Data-aware and compute-aware tasks. These works offer valuable insights that we leverage and extend in our LSN-based approach. Finally, our approach is novel in that it creates control-knobs that can be used to provision Hadoop disk I/O speed, capacity, and interconnect bandwidth to match the needs of applications in an economical fashion.

2.2.2 Efficiency in Hadoop storage

There has been extensive previous research in increasing storage and energy efficiency as well as overall throughput of Hadoop. Research on increasing the storage efficiency in GFS [111] and HDFS managed clusters [112] propose to asynchronously compress the replicated data down to RAID-class redundancy when it does not need the performance benefits of replication. However, these techniques lower MTBF, which results in lower availability and reliability. Porter [100] proposes consolidating Hadoop node disks into a SuperDataNode and running DataNode instances in virtual machines on that node. The work decouples storage from computation, but at the cost of reduced throughput and fault tolerance.

Much of the recent work focuses on energy efficiency in Hadoop storage [113] [114] [115] [116]. Rini et al. [113] propose energy aware date placement, where unpopular data is placed in a subset of Hadoop cluster nodes, generating significant periods of idleness to operate in a high-energy-saving mode without affecting nodes containing the hot data. This framework increases the skewness in popularity as hot data is concentrated on a subset of nodes, resulting in degraded throughput compared to spreading the hot data throughout the entire cluster. Amur et al. [116] and Leverich et al. [115] propose maintaining a primary replica of Hadoop on a subset of nodes that are guaranteed to be in active power state, while other replicas of the data are maintained on secondary set of nodes that are in low power modes. These energy efficiency approaches are based on an underlying assumption that the cluster is always over-provisioned in terms of number of nodes, so that the Hadoop jobs are not affected by the shrinking number of active compute nodes, which may not always be the case.

2.2.3 Tiered Storage in Hadoop

Several recent projects [117–121] focus on tiered storage for general purpose enterprise computing, mainly due to its ease of management and business value. These systems typically employ SSD based tiering and caching, along with data management across tiers, to get higher I/O rates than just from HDDs. In hat$S$, we aim to extend such storage solutions to beyond individual nodes and servers, and into Hadoop's distributed setting. The recent HDFS-2832 [122] also calls for enabling support for heterogeneous storage in HDFS. hat$S$ offers such support as well as provide different storage and data retrieval schemes to exploit the heterogeneity.

Hybrid HBase [123] explores the feasibility of introducing flash SSDs for HBase, but it only stores intermediate HBase data on the SSDs. Similarly, Spark [124] aims to avoid expensive HDD accesses by providing primitives for in-memory MapReduce computing. MixApart [110] reduces the cost and inefficiencies of shared storage systems by offering a single consolidated storage back-end for managing enterprise data and servicing all types of workloads. AptStore [107] is a federated file system for Hadoop with a unified view of data from a multitude of sources, but stores all replicas of a file on one type of device. These works share with hat$S$ the focus on using tiered storage, but differ in that they do not support storage heterogeneity.

In the distributed setting, Zebra [104], GPFS [105], and Panasas [106], offer techniques to improve read and write throughput. Similarly, Flat Datacenter Storage [108] that employs an advanced network topology, and Camdoop [125] that uses a directly-connected network topology, argue the need for better provisioning and utilization of the disk bandwidth in the modern big data applications. These works are complementary to our design. Significant research has been done on network provisioning for Hadoop but incorporating fast storage technologies along with the traditional disks has not been previously explored.

2.2.4 Storage Optimization in Hadoop

Scarlett [43] proposes a workload based scheme to increase the throughput by replicating files based on their access patterns. One shortcoming of the proposed approach is that larger files are given a priority for increased replication over smaller files, and hence popular small files may still suffer read throughput degradation and popularity skewness. Additionally, a minimum of three replicas of each file are kept regardless of their popularity, thus reducing the overall storage efficiency of the cluster. To achieve similar goals Yahoo proposes

HotROD [126], an on-demand replication scheme, which allows access to the data from some other HDFS cluster by creating a proxy node, thereby taking advantage of a replica of the same data from another data source. However, this approach might suffer degraded performance if inter-cluster network bandwidth is low. SCADS Director [127] proposes a control framework that reconfigures the storage system on-the-fly in response to workload challenges using performance model of the system. LoadAtomizer [128] propose a locality and I/O load aware task scheduler to achieve high throughput.

2.3 Heterogeneity-aware Scheduling

There has been work [129–131] on hardware-heterogeneity-aware workflow scheduling for High Performance Computing (HPC) workloads. Heterogeneity-aware scheduling algorithm have demonstrated improved performance over a heterogeneity-agnostic scheduler, multicore and many-core processors. 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.

Xie et. al. [132] proposed a data placement policy for heterogeneous compute substrate. Before execution of the jobs, a small portion data is used to run the test to measure the heterogeneity of computing node. Each node is allocated with a proportional amount of data map tasks and during the processing phase, which considers how to satisfy the data locality in the heterogeneous cloud computing environment and propose appropriate placement [133]. Chen et. al. [134] present Ussop, a grid-enabled MapReduce framework. Ussop introduces two task scheduling algorithms, Variable-Sized Map Scheduling (VSMS) and Locality-Aware Reduce Scheduling (LARS). VSMS dynamically adjusts the size of map tasks according to the computing power of grid nodes. While LARS minimizes the data transfer cost of exchanging the intermediate data over a wide-area network. LATE [98] is proposed to optimize the task scheduling based on the performance decreasing problem that generated by the default scheduling method of Hadoop in heterogeneous environment. For faster computation, a straggler, a node performing poorly though available, invokes MapReduce to run a speculative copy of the currently running task on another machine. Google claims that the speculative execution improves the job response time by 44%.The LATE algorithm actually aims to address the problem of how to robustly perform speculative execution to maximize performance under heterogeneous environment.

There are several log analysis tools [135–139] and runtime analysis tools [140–143], which can be used for performance prediction of generic distributed systems. Unfortunately, these tools do not consider MapReduce or other DFS heuristics. Recent research [23, 24, 144–147] has explored monitoring, analysis and modeling performance of individual DSF tasks with emphasis on different components of various DSFs. However, these tools do not influence the scheduling decisions based on the underlying hardware. There are a large number of software tools available for simulating distributed environments in general [148–153]. However, few Hadoop simulators exist in the proposed design space. Hammoud *et al.* [154] have simulated shared multi-core CPUs, HDD and network traffic with features like memory buffers, merge parameters, parallel copy and sort parameters. Mumak [155] and Cardona et al. [156] have implemented a simulator for MapReduce jobs, which focuses on simulating map and reduce functions as well as HDFS [157] for studying scheduling algorithms. However these scheduling decisions are not influenced by the underlying hardware.

2.4 Hadoop Workflow Scheduler

Workflows have become an integral part of modern Hadoop applications, and are managed by workflow managers such as Apache Oozie [87] and Nova [88]. 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.

Several recent works [158–160] integrate workflow management in Hadoop. Apache Oozie [87] 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 [161] supports a data flow paradigm that transforms the user generated data flow logic into Hadoop jobs. Similarly, Clustera [162] extends Hadoop to handle a wide variety of job types ranging from long running compute intensive jobs to complex SQL queries. Nova [88] workflow manager uses Pig Latin to deal with continually arriving data in large batches using disk-based processing. Kepler+Hadoop [89] is another high-level abstraction built on top of Hadoop,

which allow users to compose and execute applications in Kepler scientific workflows. Percolator [163] 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 are several other state-of-the-art projects — such as Flamingo [164], Azkaban [165] and and Nova [166] — which also provide workflow management for complex applications, and in that share their objective with our work. However, Derby*hat*S offers several key features that add value and improve the effectiveness of workflow management. First, Derby*hat*S seamlessly considers heterogeneous as well as specialized resources, and employs affinity-based job scheduling to match tasks with appropriate components for a plethora of data management DSFs. Thus, Derby*hat*S is able to go beyond managing of performance variance and truly handle emerging hybrid CPU/GPU/accelerator/disk resources (e.g., Titan [167]) as well. Second, while existing managers try to simplify the complex workflow of the modern applications by treating them as directed acyclic graphs and resolve (mostly statically) the ordering through job dependencies, Derby*hat*S uses dynamic simulation-based predictions to adjust resource allocation at runtime. This is valuable for emerging DSFs that are not merely MapReduce tasks, but rather involve iterative processes. Third, Derby*hat*S's ability to automatically handle heterogeneity will also support emerging hardware such as embedded devices. Overall, Derby*hat*S offers a holistic approach that goes beyond just workflow management by extracting and employing application-specific information through the entire application cycle. Derby*hat*S will leverage the outcomes of our three research tasks to better utilize heterogeneous resources in an application-attuned fashion – features that are desirable but currently missing from workflow managers.

2.5 Application and Workloads

In this section, we describe the representative applications — chosen from well-known Hadoop/MapReduce-based works [168–171] — that we have used in our study. We also synthesize applications based on publicly available aggregate information from production Hadoop workload traces [172, 173].

2.5.1 Basic Benchmarks

The applications in this category perform basic operations such as searching and sorting, and provide means for establishing the viability of our approach.

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. *TeraGen* is also a map-only Hadoop application that does not have any input, but writes a large output consisting of fixed-size records.

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.

Grep: Searches for all occurrences of a pattern in a collection of documents. Each mapper reads in a document, and runs a traditional "grep" function on it. The output size depends on the number of occurrences and can range from zero to the size of the input. A reducer in *grep* is simply an identity function, so in Hadoop's terminology this is a map-only application.

TeraSort: Performs a scalable MapReduce-based sort of input data. *TeraSort* first samples the input data and estimates the distribution of the input by determining r-quantiles of the sample (r is the number of reducers). The distribution is then used as a partition function to ensure that each reducer works on a range of data that does not overlap with other reducers. The sampling-based partitioning of data also provides for an even distribution of input across reducers. A mapper in *TeraSort* 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.

NutchIndex: is representative of a large-scale search indexing system. Nutch is a subsystem of the Apache search engine [174], 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.

Grep: Searches for all occurrences of a pattern in a collection of documents. Each mapper reads in a document, and runs a traditional "grep" function on it. The output size depends on the number of occurrences and can range from zero to the size of the input. A reducer in *grep* is simply an identity function, so in Hadoop's terminology this is a map-only application.

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 [175]. Bayes implements the training module for the naive Bayesian knowledge discovery algorithm atop Hadoop.

HiveBench: is representative of analytic querying on Hive [176], 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*.

2.5.2 Application Benchmarks

The applications in this category represent real workloads that are often employed in data analytics applications running on production Hadoop clusters.

Join: Performs a database join on two tables. The mappers work on rows of the tables, find a join key field (and other fields as needed), and emit a new key-value pair for each join key. After the shuffle and sort phases, records with the same join key are forwarded to the same reducer. The reducers then combine rows from the two input tables, and produce the results.

Aggregate: Performs an aggregate query on a database table. For example:

$SELECT\ b,\ sum(a)\ FROM\ table\ GROUP\ BY\ b$

The mapper reads in a row, and keeps a partial sum of a for each b, and eventually writes out b and the corresponding sum of a. A reducer will receive b and a list of partial sums of a, which it can combine to produce the final result *(b, sum(a))*.

Inverted Index: Calculates the inverted index of every word that appears in a large set of documents, and is a critical step in a web search workflow. The input data are a set of documents, each identified by a *docid*. The mapper reads in a document, scans through all the words, and outputs *(word, docid)* pairs. Shuffling and sorting merges *docid*'s associated with the same word into a list, so the reducer is simply an identity function that writes the output.

PageRank: Iteratively calculates representative score for each page, P, by adding the scores of all pages that refer to P, and is another key component of a web search workflow. The mapper reads in a record with a *docid* of a page X, its current score and *docid*s that X links to. The mapper then calculates the contribution of X to every page it points to, and emits *(docid, score)* pairs, where *docid* represent different pages and *score* is the contribution of X to those pages. The reducer reads in a *docid* for a page P', with contributions to P' from all other pages (X's), adds them together and produces the new score for P'. This process is then applied iteratively until all scores converge. For our tests, we consider only one *PageRank* iteration.

2.5.3 Trace-Based Synthesized Benchmarks

The benchmarks in this category are synthesized using models extracted from production Hadoop traces [172, 173], and help us overcome the lack of available production Hadoop workload traces for open public use. We use these applications to test our approach under realistic enterprise workloads.

Small: Emulates Hadoop tasks with an input data size of 100 KB and output limited to 1 MB, lasting for a small duration. These could be maintenance tasks, status probes, or jobs that tweak output of large applications for specific needs. The motivation behind this application is the observation that most popular applications on a Hadoop cluster consist of small jobs [172, 173].

Summary: Summarizes or filters large input data into significantly smaller intermediate and output results. For instance, the ratio between the input and final output can be as

high as 7 orders of magnitude. Such jobs are also observed in recent studies [173]. However, *summary* consists of both map and reduce phases and in that differs from *grep*.

Compute: Models the use of MapReduce in supporting large-scale computations, such as advanced simulations in Physics or Economics. The main property of this application is that cycles/byte for both mappers and reducers are about two orders of magnitude higher than the other applications we consider, thus *compute* is a CPU-bound process that produces very light I/O activity.

In addition to the above benchmarks, we also use the PigMix [177] benchmark suite for our real testbed experiments. PigMix is a popular benchmark that consists of a set of 17 queries performed on the data sets generated by the default PigMix data generator. Each query has several intermediate phases and evaluates the performance gap between Pig [178]-based execution and the direct use of Java MapReduce.

Chapter 3

Efficient Integration of Shared Storage in Hadoop

Hadoop clusters are typically built using commodity machines (nodes), and each Hadoop node typically serves as both a compute node, executing the assigned application tasks, and a storage node, storing the associated data on local disks. Any interactions between nodes occur explicitly and only during the shuffle phase of MapReduce. The application tasks spend the majority of their time using only node-local resources and consequently the system can achieve very-high scalability. This shared-nothing property also provides simplified overall application semantics; a node failure does not affect other nodes, and the failed node can be easily replaced by assigning its tasks to a different node [90].

The flip side of isolating resources in Hadoop is that idle resources at one node cannot be used to serve the needs of another node that may be experiencing a workload spike. This is critical, as although MapReduce workloads are theoretically distributed equally among participating nodes, in reality, skew in data/task assignments result in load imbalance [179, 180]. Thus, the current approach leads to inefficiencies and resource-fragmentation with nodes under- or over-provisioned and unable to support the assigned workload, even when sufficient resources may be available in the system.

Given the importance of high-performance I/O in sustaining modern big data applications, we focus on the storage performance and provisioning in Hadoop clusters. One solution used in large-scale setups is to equip each node with more disks and stripe data across them to handle load spikes [95]. However, this not only results in costly over-provisioning, but also increases node failure recovery times and complicates fault tolerance semantics. Another solution is to equip each node with an advanced storage device such as an SSD or a PCIe-based storage device. This approach is promising as even a single device can provide very high throughput, but the price-point for such devices is impractical especially for large clus-

ters — currently the cost is \$1.8/GB for a SDD [181] and \$3/GB for a PCIe-based storage device [182] compared to \$0.11/GB for hard disks, and this price gap is expected to stay large in the foreseeable future [183]. Both of the above solutions while providing the desired peak throughput during access bursts, fail to address the underlying problem of low average resource utilization.

We posit that aggregating and sharing resources across a subset of nodes can produce an efficient resource allocation in an otherwise shared-nothing Hadoop cluster. To address this, we present a novel enhancement for Hadoop, which divides a traditional Hadoop rack into several sub-racks, and consolidates the disks attached to each of the sub-rack's compute nodes into a shared Localized Storage Node (LSN) for servicing the sub-rack. The scope of a single LSN can range from serving a few nodes to perhaps a complete Hadoop cluster rack depending on workload and usage characteristics.

An observation that makes the proposed approach viable is that in typical Hadoop clusters, accesses to disks are often staggered in time because of the mix of different types of jobs and data skew across nodes. This, coupled with the bursty node workload, implies that contention at the LSN from its associated nodes is expected to be low. Therefore, by simply re-purposing a sub-rack's node-local disks in an LSN, each node can receive a higher instantaneous I/O throughput provided by the larger number of disks in the LSN. Conversely, the LSN can service its associated nodes at the default I/O throughput (experienced by nodes when using their local disks) with fewer number of disks at the LSN. We note that we do not argue for provisioning LSNs in addition to the node-local disks, rather placing some or all of the disks from a sub-rack's nodes at their LSN. Of course, moving the disks away from a node and into a shared LSN results in loss of data locality, so achieving higher I/O throughput depends on appropriate provisioning of both disks and network bandwidth to the LSN. Our design, thus, *provides a practical control knob for realizing a desired performance-cost operating point for a Hadoop cluster.*

Another advantage of LSN-based design is that it decouples storage and compute provisioning, and allows for scaling up storage to meet the demands of big data applications by simply provisioning more disks at the LSN. Consolidating data into fewer high-density nodes opens the door for a myriad of global decisions and optimizations, such as deduplication, compression, and snap-shot generation. Standard enterprise fault tolerance techniques, such as RAID-5 and RAID-6, can also be employed more easily in consolidated storage such as that envisioned by our LSN-based design.

3.1 Integrating Shared Storage In Hadoop

In this section, we first motivate our design of grouping several Hadoop nodes into a sub-rack and consolidating the nodes' direct attached disks into a local shared storage to service the sub-rack. Then, we outline several alternative shared-storage designs.

3.1.1 Rationale and Motivation

Application data sets continue to grow at unprecedented rates. To keep up with this trend, the per-node disk capacity on Hadoop clusters is increasing rapidly, e.g., from two 80 GB disks in the original MapReduce deployment [111] to four and (even eight) 3 TB disks [95] in modern Hadoop setups. This raises several issues about the viability of using node-local storage for all data. First, DAS-based Hadoop architecture has the limitation that the storage capacity is tightly coupled with compute capacity. To add more storage, more compute nodes need to be added. However, the cost of the extra compute power is unnecessary for the typically I/O-bound Hadoop applications. Second, simply adding more disks to local nodes increases the chance of some disks failing, and reduces the already typically low Mean Time Between Failures (MTBF) of a Hadoop node. Moreover, this option increases the direct acquisition costs, e.g., more disks and related hardware has to be bought, as well as the indirect maintenance and operating costs, e.g., more disks would consume more energy. Third, when using commodity Hadoop hardware, a significant time and bandwidth resources are spent on recreating replicas after node or disk failures. RAID on local disks can help mitigate this. However, per-node RAID creates overhead on both capacity and performance. For instance, assuming a per-node RAID-5 configuration with 4 data and 1 parity disks, the capacity and parity read/write overhead on write accesses is a very high 20% on every node in the cluster. A promising solution in this context can be realized by using Enterprise storage servers that can offer lower MTBF than commodity hardware clusters and ensure lower failure rate for the data [184,185]. Fourth, provisioning all the storage needs of a node locally prevents the use of advanced devices, such as SSDs, as current price-points make it economically impractical to deploy such approaches at all the nodes. Finally, we argue that following the conventional wisdom of treating data-locality as the only design constraint in Hadoop clusters, results in a sub-optimal setup, both in terms of performance and efficient utilization of resources. Factors such as storage utilization can no longer be ignored in the face of growing data sets.

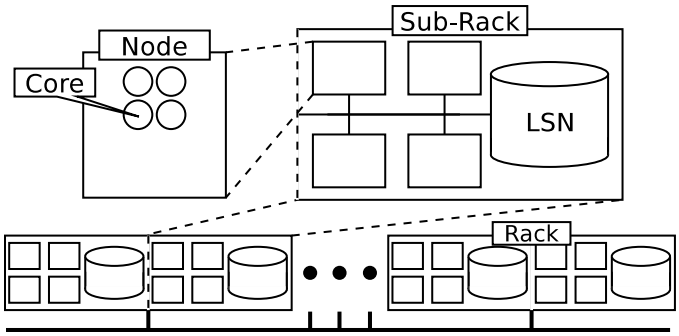


Figure 3.1 Hadoop architecture using an LSN.

Consider the following scenario. If a single Hadoop task utilizes a single disk for 5% of its execution time, ideally 20 tasks would be needed to fully utilize a single disk, given that tasks are all staggered so they do not compete with each other. Similarly, if a node has four local disks, it would take 80 tasks running concurrently to fully utilize all disks on the node. Moreover, if the tasks are not uniformly distributed across nodes, which is typically the case, there will be a skew in the load with some nodes experiencing I/O bottlenecks, while others with idle disks. Thus, the imbalance due to the node-local disks serving only their associated nodes leads to resource fragmentation. We argue that by consolidating disks from a group of nodes into an associated localized storage node (LSN), the resources unused by a lightly-loaded node can be used by a heavily-loaded node, consequently avoiding resource fragmentation and providing for better storage utilization.

The flip side is that by moving disks from nodes to LSN, the approach sacrifices some data locality. However, it has been observed that the loss of locality can be mitigated to some extent by better provisioning of the interconnect bandwidth [102, 144]. To this end, we propose consolidating the disks from a small number of compute nodes into an LSN. This approach has the potential to offer higher average disk utilization, simplified management of data, and reduced replica recreation by making it viable to employ RAID and advanced storage solutions at the consolidated LSNs.

3.1.2 Storage Sharing Scenarios in Hadoop

A consolidated shared storage system can reside at different levels of the Hadoop architecture. In the following, we present three potential alternative scenarios for sharing storage in Hadoop.

3.1.2.1 Naive Storage Consolidation

A first cut design is to take all MapReduce related data and move it to a single consolidated storage outside the entire Hadoop compute cluster, and provision a very high bandwidth link between the compute nodes and the storage system. Such a setup is often deployed to connect a cluster file system to a supercomputer [186–188], and has also been proposed for Hadoop by recent commercial offerings [189–191]. However, in this configuration a typical large-scale data-intensive Hadoop application would create an almost constant high-volume I/O flood to the storage system, which would quickly saturate the storage connection link and become a bottleneck. Moreover, aggregating storage cluster-wide would require a sophisticated cluster file system that treats the storage nodes as an integrated unit. This in turn would entail complexity in managing failures and providing high performance. Moreover, the intermediate data traffic, i.e., during the shuffle phase, can be orders of magnitude higher than the input and output data. A cluster-wide consolidated storage-to-compute interconnect can quickly become overwhelmed and lead to unacceptable performance degradation. Consequently, such a design goes against the very spirit of the MapReduce model that achieves unprecedented scalability by treating the cluster resources as loosely coupled with locally stored data, which are readily replaceable.

3.1.2.2 Localized Storage Consolidation

The main bottleneck in the previous case is the interconnect between the global shared storage and Hadoop nodes. In our next approach, shown in Figure 3.1, we limit the number of compute nodes that share a storage system, i.e., a sub-rack whose size range from a fraction of a rack to perhaps a complete rack. We refer to the shared storage as Localized Storage Node (LSN). The intuition behind this local consolidation approach is that it avoids the bandwidth bottlenecks by limiting the sharing to a few nodes instead of the whole cluster. All the disks from the sub-rack's compute nodes are consolidated into a corresponding LSN for the sub-rack. The LSN supports both HDFS and shuffle data for the sub-rack.

In this configuration, map tasks no longer have node-level locality and must retrieve data from the corresponding LSN in the sub-rack. However, since data is now striped across a larger number of disks at the LSN than those of a single node, the LSN can provide much higher I/O throughput, which can mitigate the impact of lost locality. Moreover, since only a small number of nodes share an LSN, only the inter-rack interconnect is used for

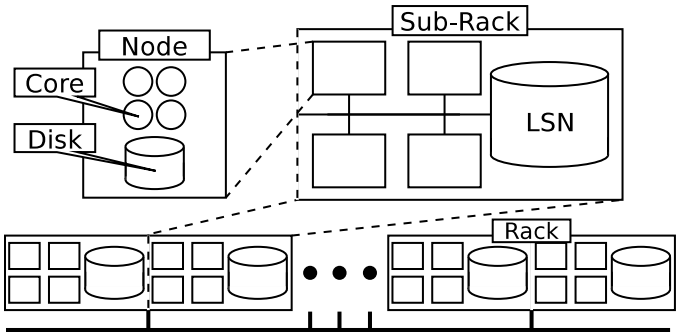


Figure 3.2 Hadoop architecture using a hybrid storage design comprising of a small node-local disk for shuffle data and an LSN for supporting HDFS.

accessing data, and multiple sets of nodes (in different racks) can interact with their LSNs simultaneously, avoiding a global bottleneck.

3.1.2.3 Hybrid Storage Consolidation

A problem faced in the previous design is that each compute node requires at least one local disk to run its operating system and temporary data, and thus makes it impractical to remove all disks from a node to its associated LSN. The key insight of our next design, shown in Figure 3.2, is to buffer shuffle data, which is not replicated and usually consumed shortly after it is generated, on the node-local disk. Thus, we design a hybrid setup where the LSN stores HDFS data for a sub-rack of nodes, while a node-local disk buffers shuffle data and stores OS files required to run the node.

An additional advantage of the hybrid approach is that it paves the way for economically incorporating SSDs in the Hadoop architecture. For instance, the node-local disks can be replaced by (low-capacity) SSD devices for holding the OS and serving as a buffer for in-memory shuffle data. Given the good random I/O (especially read) performance of SSDs [192], handling shuffle data would be a well-matched use-case for them. This is also advocated by recent work on the importance of memory-locality rather than disk-locality in Hadoop [102].

Based on the above observations, we adopt the hybrid LSN approach in our design. This would allow for efficiently integrating economical large-capacity enterprise filers, as well as advanced storage devices such as SSDs into the Hadoop ecosystem. Our approach also helps to sustain Hadoop storage subsystem in the face of the growing data deluge created by modern applications.

Table 3.1 Representative MapReduce (Hadoop) applications used in our study.

Application	Map		Reduce	Number		Cost (cycle/byte)	
	Input	Output	Output	Mapper	Reducer	Map	Reduce
Grep	10 GB	1 MB	1 MB	160	1	40	10
TeraGen	0 KB	10 GB	–	40	–	10	–
TeraSort	10 GB	10 GB	10 GB	160	40	40	10
Join	10 GB	1 GB	10 MB	160	40	400	100
Aggregate	10 GB	100 MB	10 MB	160	10	40	20
Inverted Index	1 GB	10 GB	100 MB	40	40	40	10
PageRank	1 GB	10 GB	1 GB	40	40	100	20
Small	100 KB	1 MB	10 KB	4	1	400	100
Summary	10 GB	10 MB	10 KB	160	1	40	10
Compute	1 GB	10 GB	100 MB	40	40	4000	1000

3.2 Evaluation of Shared Storage In Hadoop

In this section, we present the evaluation of the hybrid localized shared storage design in Hadoop. We compare the *baseline* Hadoop to our LSN-based implementation using simulations as well as extensive experimentation on a medium-sized cluster. We perform our experiments using representative applications — chosen from well-known Hadoop/MapReduce-based works [168–171]. We also synthesize applications based on publicly available aggregate information from production Hadoop workload traces [172, 173]. Table 3.1 lists the applications, and for each also summarizes parameters such as the input and output data size, the number of mappers and reducers, and the compute-cost of map and reduce tasks, which we use in our simulations.

3.2.1 Tests Using a Real Cluster

Our first set of tests explore the impact of LSN on Hadoop performance using a real cluster.

Experimental Setup Our testbed consists of a master node and 21 worker nodes serviced by three LSNs. The nodes have two 2.8 GHz quad-core Intel Xeon processors, 8 GB of RAM, and one SATA disk. The LSN nodes are identical to the rest of the nodes, but contain five SATA disks. The disks are Seagate Barracuda ES.2 7200 RPM with 500 GB capacity. The machines are connected to a dedicated Gigabit switch via 1 Gbps links as well as a dedicated InfiniBand switch via 10 Gbps links. In the experiments, only one of the network interfaces

Table 3.2 Specification of networks and disks used in the testbed.

	Network		Data Storage		
Conf	N_1	N_2	D_1	D_2	D_3
Nodes	1 Gbps	10 Gbps	1 disk		
LSN	1 Gbps	10 Gbps	1 disk	3 disks	5 disks
Speedup	1	10×	1	3×	5×

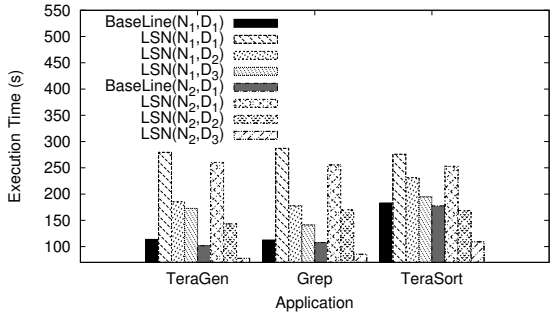


Figure 3.3 Comparison of Hadoop (*baseline*) with LSN-based configurations.

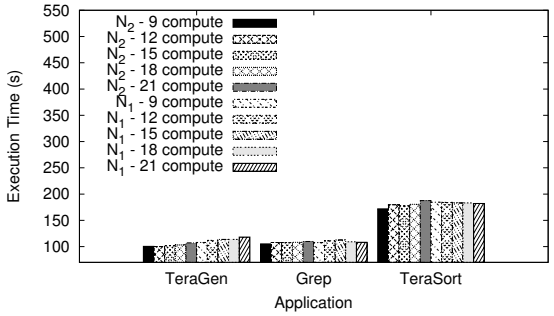


Figure 3.4 Comparison of Hadoop (*baseline*) with increasing number of compute nodes.

is used at a time. Each worker node is configured with six map slots and two reduce slots to ensure that all the available cores on the node are utilized. The considered benchmarks are mostly map intensive, so there are more number of map slots than the reduce slots.

To realize a localized storage node within Hadoop, we configured TaskTrackers on each worker node as is done in standard setups, but configured the DataNode to run on the LSN. Although we did not initialize a TaskTracker on the LSN, our system does not prevent it, and it is an option we plan to investigate in future work.

We ran the three basic benchmarks : *TeraGen* with one mapper per compute node, *Grep* and *TeraSort* each with 16 mappers and two reducers per compute node. *TeraGen* generates 1 GB of data per worker node, which is the input for *Grep* and *TeraGen*. The master node runs both the Hadoop JobTracker and NameNode for all the experiments. The LSN nodes provide storage for the shared storage experiments. For all studied cases, the shuffle data is stored on the node-local disks as described in section 3.1.2.3. All cases use a HDFS replication factor of three. We study several different testbed configurations by varying both the network provisioning and the number of disks at the LSN. These configurations are shown in Table 3.2. Finally, the numbers reported represents averages of five different runs, and little variance was observed between runs.

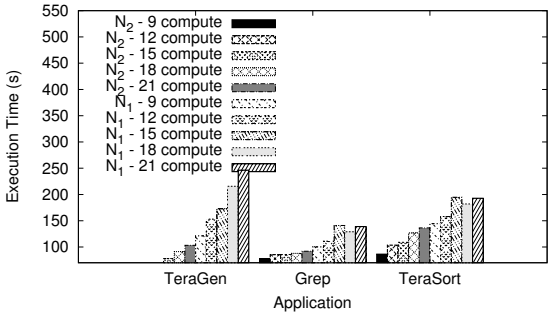


Figure 3.5 Comparison of Hadoop (*LSN*) with increasing number of compute nodes.

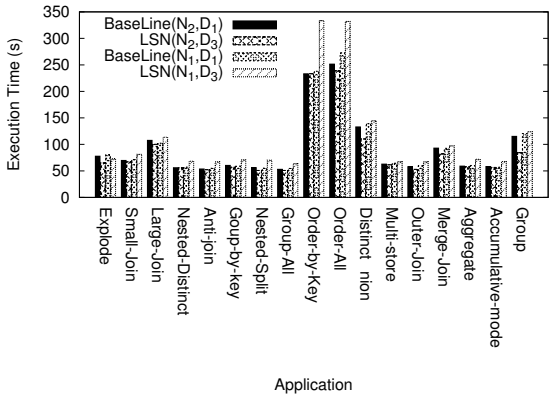


Figure 3.6 PigMix benchmark execution times under Hadoop (*baseline*) and LSN-based configurations.

LSN Performance For our first experiment, we used 15 of the available worker nodes to compare the performance of standard Hadoop (*baseline*) to that of LSN-based Hadoop using different network and storage provisioning. Figure 3.3 shows the results for six hybrid configurations with three LSNs, one per five worker nodes, and two standard configurations, where (N_x, D_y) refers to N_x and D_y in Table 3.2. Consider $LSN(N_1, D_1)$, which consolidates all the HDFS storage onto a single remote disk per LSN, without changing the network bandwidth. Not surprisingly, going from 15 local disks to three remote disks slowed *TeraGen*, *grep* and *TeraSort* by 2.45×, 2.55× and 1.50×, respectively, compared to $Baseline(N_1, D_1)$. The three disks do not have enough bandwidth to keep up with the 15 compute nodes and hence become a bottleneck.

Next, we test $LSN(N_2, D_1)$, which increases the network throughput available to each node, but still has only one disk per LSN. The higher network bandwidth does not improve the performance and *TeraGen*, *grep* and *TeraSort* execute 2.55×, 2.37× and 1.42× slower than $Baseline(N_2, D_1)$, respectively. Next, we increase the number of disks, but keep the network bandwidth as the baseline in $LSN(N_1, D_2)$. We see that the addition of disks improves the performance over $LSN(N_1, D_1)$ by 1.5×, 1.6× and 1.2× for *TeraGen*, *grep* and *TeraSort*, respectively. The $LSN(N_1, D_3)$ configuration uses even more disks, however, here the network becomes the bottleneck there is little extra benefit observed. Finally, we test better provisioning of both network and storage, which is the intended deployment of our system. $LSN(N_2, D_3)$ sees a performance increase of 24%, 20% and 39% for *TeraGen*, *grep*, and *TeraSort*, respectively, compared to $BaseLine(N_2, D_1)$.

These results show that balancing shared disk provisioning with an adequate network throughput to the LSN can perform better than the shared-nothing *baseline* Hadoop.

LSN Scalability In our next experiment, we examine the scalability of the LSN-based design by varying the number of compute nodes that are serviced by a single LSN. In order to show an accurate performance comparison between the configurations, we keep the amount of data processed by each compute node constant. Figure 3.4 shows that in standard Hadoop, as expected, keeping per node data constant does not significantly affect the execution time with increasing number of nodes.

Figure 3.5 shows that increasing the number of compute nodes serviced by an LSN impacts the performance significantly as the size of the workload increases proportionally as well. We see that when three $LSN(N_2, D_3)$ serve nine compute nodes, they perform better than the *baseline*(N_2, D_1) by 43%, 26% and 50% for *TeraGen*, *grep*, and *TeraSort*, respectively. In this case the *baseline*(N_2, D_1) configurations use nine disks while the $LSN(N_2, D_3)$ uses 15 disks.

In our next experiment, we provision three $LSN(N_2, D_3)$ to serve all of the available 21 compute nodes. We see that in spite of disk reduction from 21 in *baseline*(N_2, D_1) to 15 in $LSN(N_2, D_3)$, the later performs 14% and 23% better in *grep* and *TeraSort*, respectively, while *TeraGen* performs similarly for both cases. Figure 3.5 also shows performance results with varying number of compute nodes with a similar pattern. We also see that, when nodes are provisioned with lower network bandwidth, *baseline* performs better because of the network contention at the LSNs.

This test shows that by consolidating disks into LSNs, we can either support the same throughput with fewer disks, or support higher throughput with the same number of disks. This allows the cluster designers to select a configuration most suited to their application needs.

Compute Intensive Workloads *TeraGen*, *grep*, and *TeraSort* spend significant time on I/O in HDFS. For our next test, we observe the behavior of LSN under shuffle intensive and compute intensive workloads using the PigMix [177] benchmark suite. Figure 3.6 shows the performance results under different Hadoop configurations. We see that *baseline*(N_2, D_1) and *baseline*(N_1, D_1) perform 8% and 11% slower than $LSN(N_2, D_3)$, respectively. The variation in the performance between different queries is due to the variation in I/O intensity of each query. LSN does not have a significant effect on the performance of compute intensive and shuffle intensive workloads, but positively influence I/O intensive ones.

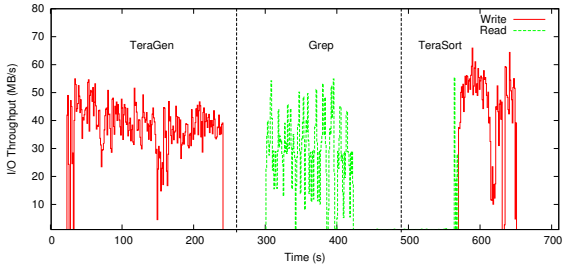


Figure 3.7 Disk I/O throughput observed at one node in *baseline*. Other nodes exhibit similar patterns.

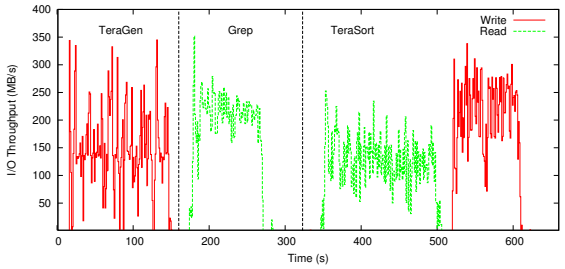


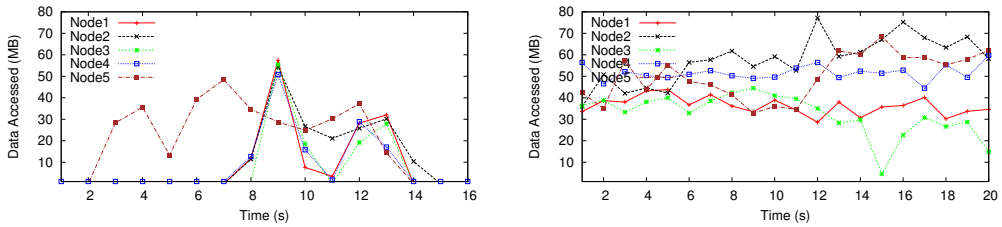
Figure 3.8 Disk I/O throughput observed at the LSN.

Disk Bandwidth Utilization In our next experiment, we examine in detail the disk bandwidth utilization under *baseline* and the hybrid LSN. To perform the rest of the experiments, we consider five worker nodes, one master node, and one LSN. We also increase the overall data generated by each node to 2 GB.

Figure 3.7 shows the disk I/O throughput observed at the local disk on one of the worker nodes under *baseline*, when the applications are run in a sequence. We observe that the access pattern is bursty in nature, achieving disk bandwidth utilization of 45%, 32% and 21% for *TeraGen*, *grep* and *TeraSort*, respectively. The raw read and write I/O bandwidth observed for the disks are 118 MB/s and 83 MB/s, respectively. In *TeraGen* and *grep*, we see that the disk is under constant load but the bandwidth is not fully utilized. This is because of the large number of small simultaneous reads and writes in these workloads. In the case of *TeraSort* there are fewer read and write accesses with idle periods in between, which cause under-utilization of the disk bandwidth.

We contrast these results with the average bandwidth utilization of one of the disks in LSN shown in Figure 3.8. Here, average bandwidth utilization has increased to 32%, as three disks in this configuration are consolidated to handle the load of the five clients.

Workload Characteristics Next, we examine the read and write accesses from the worker nodes observed at the LSN. Figure 3.9 shows a snippet of read (*TeraSort*) and write (*Ter-*



(a) Read accesses. (b) Write accesses.

Figure 3.9 Snippets showing disk accesses from different nodes observed at the LSN.

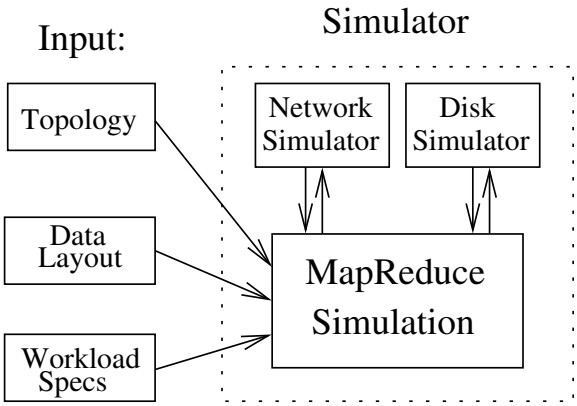


Figure 3.10 *MRPerf* architecture.

aGen) accesses. We see that workers do not issue their I/Os to the LSN at the same time even though they are running the same applications. In fact, the average, standard deviation, and maximum of the sum of instantaneous read and write throughput observed from all the workers across the applications is 6.5 MB/s, 22.5 MB/s and 64.3 MB/s, respectively. This behavior confirms our intuition that accesses from multiple nodes arrive at the consolidated LSN in a staggered manner, thus allowing for the LSN to service multiple nodes efficiently without being inundated by the combined accesses.

3.2.2 Simulating Hadoop Clusters

To explore the various aspects of our approach in detail we also employ simulations of Hadoop clusters. There is ample previous research done on modeling and simulation of MapReduce workloads and setups [144, 193–198], which we leverage. We choose our MRPerf [144] discrete event Hadoop simulator, as it has been previously used to study impact of data locality, alternative network topologies, and failure [144, 199–201]. The simulator provides us with means to investigate the performance impact of system features such as node, rack, and

network configurations, disk parameters and performance, data layout, and application I/O characteristics — all of which we want to explore in the context of our LSN design.

Figure 3.10 shows the overall architecture of the MRPerf simulator. The simulator takes as input the topology of a cluster, the parameters of a job, and a data layout, and produces detailed simulation results about how the job would behave on the specified cluster configuration. The input configuration is provided in a set of files, and processed by different processing modules (*readers*), which are also responsible for initializing the simulator. To model a specified setup, MRPerf creates a number of simulated nodes equipped with multiple processors and disks, and supports different ways to distribute the resources between the jobs scheduled for the node. The ns-2 driver module provides the interface for network simulation. Similarly, the disk module provides modeling for the disk I/O. The simulator implements MapReduce heuristics that simulate the map and reduce tasks, manage their associated input and output data, make scheduling decisions, and model disk and processor load. MRPerf offers fine-grained simulations that can capture the impact of a specific cluster configuration on application behavior at different stages of execution. For example, the network bandwidth between nodes is not important for a job that produces little intermediate output, if the map tasks are scheduled on nodes that hold the input data. However, for the same application, if the scheduler is not able to place the jobs near the data, the network bandwidth between the data and compute nodes might become the performance limiting factor. MRPerf models these interactions to correctly predict application behavior on a given Hadoop setup [144].

In this work, we extend the simulator to support our application-oriented evaluation, especially to explore cases that we cannot provision on our real cluster, such as an LSN with varying number of disks ($8-64$), varying the network provisioning of the cluster (bandwidth: $4\ Gbps - 40\ Gbps$), and provisioning each local node with a SSD.

3.2.3 Simulation Results

In our next set of tests, we use simulations to study the impact of the LSN-based Hadoop in detail. The traces used to drive the tests are deterministic and the reported numbers do not change across multiple runs.

As discussed in section 3.2.2, we use MRPerf [144] for our simulations. We focus on simulating a single sub-rack and analyze in detail a number of different LSN design choices and their

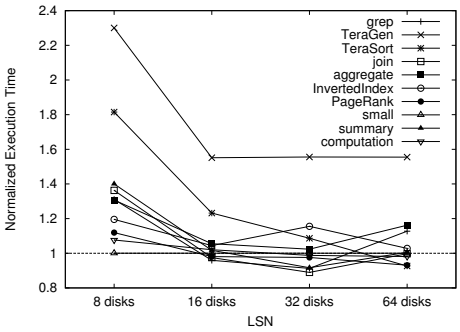


Figure 3.11 Performance of *baseline* Hadoop and LSN with different number of disks in LSN. Network speed is fixed at 40 Gbps.

impact on performance. We believe that the conclusion we draw will generalize to larger clusters comprising multiple sub-racks, each with its own LSN. This is true especially when a better provisioned communication channel is used between nodes and their associated LSNs to avoid the interconnect contention due to node-LSN traffic of different LSNs.

One LSN Servicing 20 Nodes We consider a topology with 20 nodes serviced by a single LSN. Each of the 20 nodes has eight cores and four disks and is connected via 1 Gbps links. In the LSN case, we aggregate up to 64 disks, leaving one disk at each node, and connect the LSN to the switch via a 40 Gbps link. In some of the cases, we also increase each node's interconnect to 2 Gbps links and equip them with SSDs. All experiments are run with eight map and four reduce slots.

In Figure 3.11, we change the number of disks provisioned at the LSN and report the execution time of each application normalized to the case of *baseline*. In this test, the LSN's network is set to a maximum throughput of 40 Gbps to make sure it does not become a bottleneck. The performance numbers of the $LSN(N_{40}D_{16})$ configuration are within 5.5% of the respective values for the *baseline* Hadoop for eight out of ten applications. This illustrates the efficiency of our disk aggregation technique. In this 20-node cluster, we are able to efficiently utilize 55% fewer disks ($20 + 16 = 36$ disks in $LSN(N_{40}D_{16}$ compared to $20 * 4 = 80$ disks in *baseline*) to achieve comparable performance for the studied applications. The two applications, *TeraGen* and *TeraSort*, which are very output-heavy see a 55% and 23% slowdown, respectively. In both these cases, the LSN becomes a bottleneck as it is unable to keep up with the workload.

Next, we investigate the impact of network bandwidth on application performance. For this experiment, we set the number of disks to 64. The results are plotted in Figure 3.12. We see that an LSN with 4 Gbps connection is sufficient to yield the same performance as *baseline*

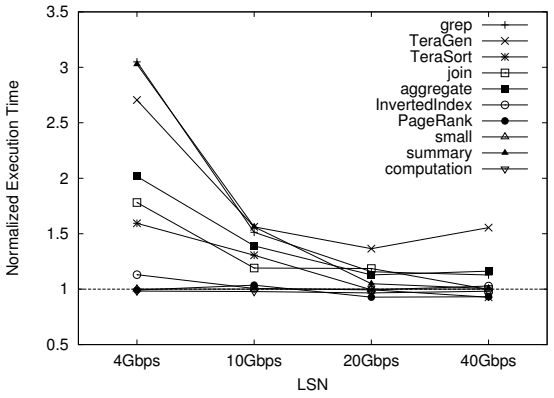


Figure 3.12 Performance of *baseline* Hadoop and LSN with different network bandwidth to LSN. The number of disks at LSN is fixed at 64. The network speed is fixed at 4 Gbps.

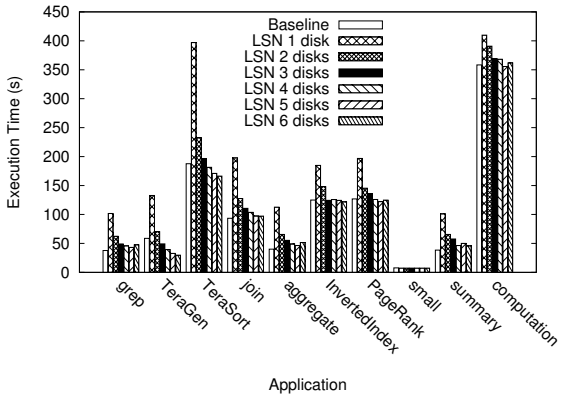


Figure 3.13 Performance of *baseline* Hadoop and LSN with different number of disks in LSN.

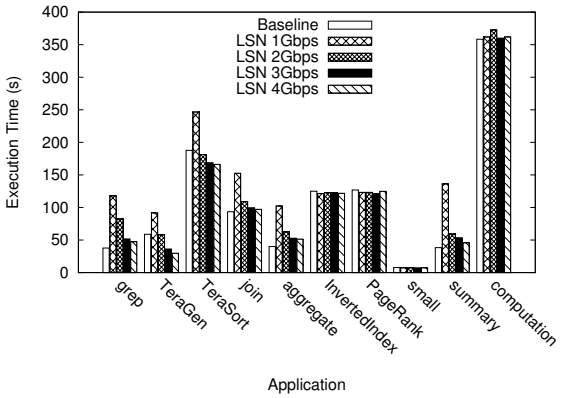


Figure 3.14 Performance of *baseline* Hadoop and LSN with different network bandwidth to LSN. The number of disks at the LSN is fixed at 6.

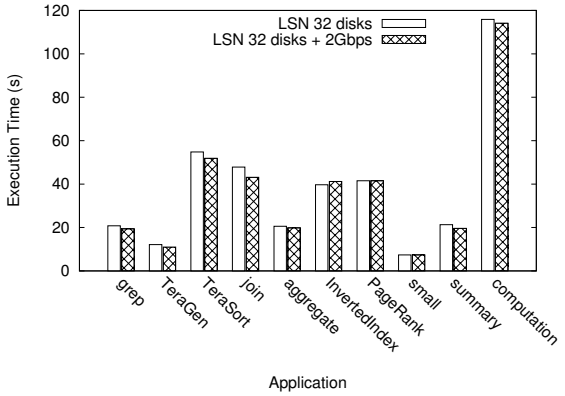


Figure 3.15 LSN performance with Hadoop nodes equipped 2 Gbps links.

for half of the applications, i.e., the ones that do not generate large amount of data. Moreover, the 20 Gbps link is enough to bring performance of all applications except *TeraGen* to within 18.6% of that under *baseline*.

5-node LSN Simulation In our next test, we set up a simulation topology with five nodes serviced by a single LSN to simulate a smaller LSN to node ratio, and again study the impact of different design choices. All nodes are connected through a single switch. The connection speed for each node and LSN is 1 Gbps and 4 Gbps, respectively. Each of the five nodes has eight cores and two disks, while the LSN has six disks. We run all the ten applications outlined in Table 3.1 and record the results. Once again, each node is configured with eight map slots and four reduce slots.

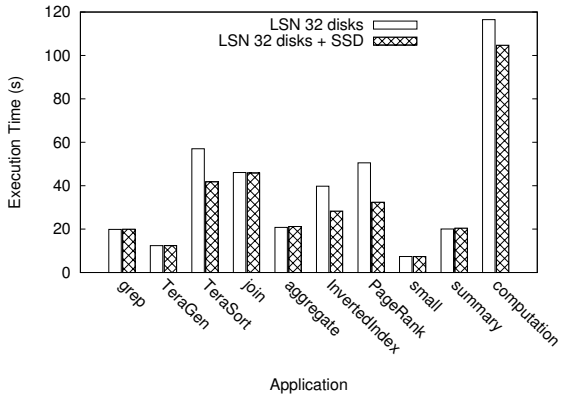


Figure 3.16 LSN performance with Hadoop nodes equipped with SSDs.

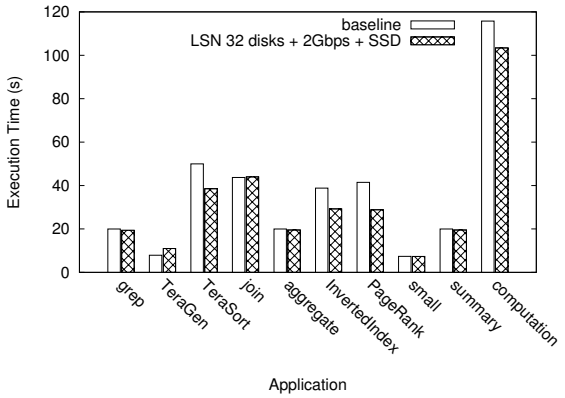


Figure 3.17 *baseline* Hadoop performance compared to LSN with nodes equipped with SSDs and 2 Gbps links.

The test studies performance under varying number of disks provisioned at the LSN. Figure 3.13 shows the results, and highlight several design trade-offs for LSN. First, LSN with four disks can match the performance of *baseline* Hadoop within 3.7%, on average, and there is almost no benefit of adding more disks. This means that the LSN provides savings of one disk (9 disks in LSN versus 10 in *baseline*). Second, output heavy jobs experience a significant performance boost, 33% for *TeraGen*, compared to *baseline* provided mainly by the reduced number of replicas. Moreover, LSN can provide load balance across multiple nodes, and achieve high overall performance. Third, read heavy workloads, such as *Grep*, *Aggregate*, and *Summary* exhibit more uniform access patterns to the local disk (Figure 3.7), and consequently experience a slowdown, 18.7% on average, when running on the LSN. This is because aggregating such accesses at the LSN does not provide an additional benefit. Finally, the performance of the rest of the applications is within 4.6% of that under *baseline*.

Next, we vary the bandwidth available at the LSN from 1 Gbps to 4 Gbps, and observe the performance impact. Figure 3.14 summarizes the results. The four applications — *InvertedIndex*, *PageRank*, *Small*, and *Computation* — that do not consume or generate large amounts of data, but rather are CPU intensive or operate on large amount of intermediate data, see no benefit from increasing the LSN's network. In contrast, the rest of the applications that do heavy input/output, experience a significant slowdown from the *baseline* under 1 Gbps link at the LSN. Provisioning 3 Gbps at the LSN, however, is enough to handle the client workload with a performance overhead within 5.7%, on average.

Better Provisioned Local Nodes So far we have examined various provisioning scenarios for the LSN. In the next set of experiments, we explore several design options at the node

side. To ensure that the LSN is not a bottleneck in these tests, we provision it with 32 disks and a 20 Gbps network and set the map and reduce slots to four.

First, we examine the impact of increased local bandwidth of each node as seen in Figure 3.15. An increase of 2 Gbps in the link bandwidth results in an average speedup of 4% across all applications; most noticeable is the 9.9% speedup for *Join* that benefits from the extra bandwidth for both its heavy HDFS and shuffle traffic.

Our hybrid LSN approach significantly decreases the number and size of disks needed to be provisioned on each node, which lets us optimize each node by replacing its hard disk with an economically viable small capacity SSD. The only workload related data that needs to be stored at the nodes is shuffle data that tends to be mostly random accesses [168]. The shuffle works in a pull model, where consuming reducers proactively retrieve data from producing mappers [202]. Hence, the workload is characterized by sequential writes and random reads, which is a good match for the excellent random read performance of SSDs. The results of adding an SSD to each node are shown in Figure 3.16. *TeraSort*, *InvertedIndex*, *PageRank*, and *Computation*, all of which process a lot of intermediate data, get a significant performance boost (25.4% on average).

Finally, we combine the node-side optimizations of using an SSD and a faster link, and compare the performance to *baseline* Hadoop. The results are shown in Figure 3.17. The optimizations together help us reduce the performance gap compared to *baseline* for even the most data intensive applications such as *TeraGen* that shows 39.3% slowdown from the earlier observed 53.7%. The rest of the benchmarks achieve a 10.7% speedup in general and prove that our hybrid localized storage is a viable augmentation of the otherwise shared-nothing Hadoop architecture.

Equipment Cost Comparison Provisioning an extra LSN machine is cost-efficient compared to equipping all participating nodes with more advanced hardware and software to handle the increasing number of disks needed to support the growing data. The cost of disks and network is more crucial for choosing our design.

In the above experiment, our approach uses 20% less disks: 5×2 per worker $= 10$ in *baseline* versus 5×1 per node $+ 3$ at the master $= 8$ for $LSN(N_2, D_2)$, i.e., savings of two disks. At the same time, we use 62.5% more network ports: 5×1 per worker $+ 1$ at the master $= 6$ in *baseline* versus 5×1.35 per node $+ 3$ at the master $= 9.75$ for $LSN(N_2, D_2)$, i.e., 3.75 extra ports based on our observed bandwidth given the overhead of port bonding, or

8 extra ports, otherwise. Assuming an average 500 GB disk costs about $50 and a network port costs $5 each, this can yield cost savings of 81% (for 3.75 ports case) and 60% (for the 8 port case). This is by no means a thorough price-point analysis, nonetheless it serves as an indication that LSN-based design can be cost-effective and deliver high performance.

In summary, these results show that our approach of consolidating local disks into an LSN, and enabling sharing in the shared-nothing Hadoop is able to achieve better disk utilization without sacrificing performance (by mitigating the locality loss).

3.3 Chapter Summary

In this chapter, we revisit the cluster architecture of Hadoop to better provision per-node storage resources in the face of growing application datasets. We observe that simply adding more disks to individual Hadoop nodes that often exhibit bursty workloads is not efficient. This approach results in low overall disk utilization, increases costs as well as the chances of node failures, and the large capacity elongates the time it would take to recreate a failed replica. Also, scaling storage coupled with compute adds extra cost of nonstorage resources that are not necessarily required for I/Ointensive workloads, and reduces overall system efficiency. To this end, we study the impact of LSN on Hadoop application performance using a range of representative applications and configuration parameters. Our evaluation shows that a single LSN servicing 20 compute nodes can achieve performance within 5.5% of standard Hadoop on average, for eight out of ten applications studied, while using a little over half (55%) of the number of disks in the standard setup. Moreover, for a case where an LSN is used by 5 compute nodes, we observe up to 12% performance improvement while using 28disks.

Chapter 4

A Heterogeneity-Aware Tiered Storage for Hadoop

A promising trend in storage technologies is the emergence of heterogeneous and hybrid storage systems [32–35] that employ different types of storage devices, e.g., SSDs, ramdisks, etc. Moreover, the networking infrastructure bandwidth is growing at a pace that is an order of magnitude higher than the I/O bandwidth improvements in hard disk drives (HDDs) [36]. The two trends are enabling realization of distributed, hierarchical, hybrid and heterogeneous storage solutions that are efficient and cost effective, e.g., Hystor [34] and ConquestFS [37]. These systems typically integrate HDDs with fast emerging storage mediums, e.g., ramdisks, SSDs, etc. The faster storage serves as a buffer for frequently accessed data and yields very high I/O rates, while the HDDs store the infrequently accessed data and provide cheap high-capacity storage for the large data volumes.

Inspite of the above developments, it is a challenge to leverage advanced storage solutions in the context of Hadoop. This is because the Hadoop Distributed File System (HDFS) [203] — that serves as the storage substrate for Hadoop clusters — is not designed to handle heterogeneous storage. HDFS treats all the underlying storage components to be comprised of blocks with same I/O characteristics. Thus data is distributed uniformly across all the storage devices, irrespective of their I/O characteristics and capacity, which leads to inefficiencies and resource wastage.

One way to incorporate emerging storage devices into Hadoop is to equip the nodes with one type of device only, e.g. SSD of the same type. However, this is impractical as the cost per GB of such devices is still far from the economical storage offered by HDDs, and this cost gap is expected to remain high in the near future [204]. Thus cluster deployments are likely to adapt the hybrid approach of using HDDs along with a variety of storage devices. Moreover, large clusters typically go trough several upgrade phases [95] during their lifetime,

thus all the nodes can not be expected to have homogeneous storage performance even if only HDDs are utilized. Yet another source of heterogeneity is the emergence of enterprise consolidated storage solutions for Hadoop [91, 92, 205, 206], which couple node-local storage with network-attached central storage to provide ease of data management while sustaining high I/O rates. Thus there is a need for enhancing the Hadoop storage layer to manage heterogeneity in the underlying storage systems.

We explore the utility of heterogeneous storage devices in Hadoop and address challenges therein by designing hat$S$, a heterogeneity-aware tiered storage for Hadoop. hat$S$ logically groups all storage devices of the same type across the nodes into an associated "tier." A deployment has as many tiers as the different type of storage devices used, and a node with multiple types of devices is part of multiple tiers. For instance, if a deployment consists of nodes with a SSD and a HDD, all the SSDs across the nodes will become part of a SSD tier, and similarly all the HDDs will form the HDD tier. By managing the tiers individually, hat$S$ is able to capture the heterogeneity and exploit it to achieve high I/O performance.

Contrary to HDFS that only considers network-aware data placement and retrieval policies, hat$S$ proposes additional policies to replicate data across tiers in a heterogeneity-aware fashion. This enhances the utilization of the high-performance storage devices by efficiently forwarding a greater number of I/O requests to the faster tier, thus improving overall I/O performances. To facilitate this, in addition to the standard HDFS APIs, hat$S$ also provides custom APIs for seamless data transfer across the tiers and management of stored data in each tier. These features allow hat$S$ to integrate heterogeneous storage devices into Hadoop to extract high I/O performance.

4.1 Design of hat$S$

hat$S$ enhances HDFS for heterogeneous storage devices by creating a storage hierarchy based on the performance characteristics of the devices, and designing heterogeneity-aware data placement and retrieval policies that improve overall I/O performance in Hadoop clusters.

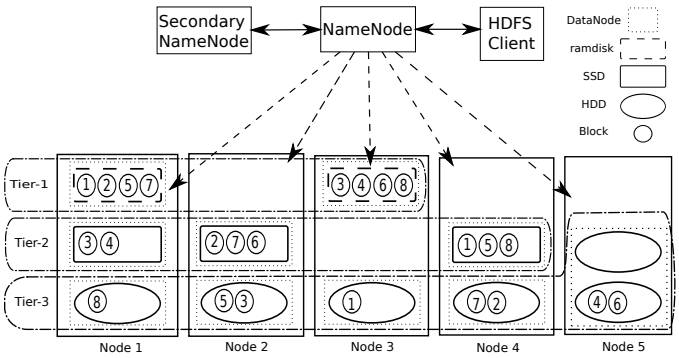


Figure 4.1 hat$S$ architecture overview.

4.1.1 System Architecture

Figure 4.1 shows the overall architecture of hat$S$. An important difference between hat$S$ and HDFS is the design of the DataNode. In HDFS, each participating node hosts a single DataNode instance, constituting multiple storage devices, regardless of their characteristics such as supported I/O rates and capacity. In contrast, each participating node in hat$S$ hosts multiple DataNode instances, where each instance represents only one type of storage device. For example, a node with two HDDs and a SSD will have two DataNodes in hat$S$, one associated with the HDDs and the other with the SSD.

All devices of the same type and similar I/O characteristics, e.g., all similar HDDs, across all the participating nodes are logically grouped into a virtual storage "tier." For example, a tier of HDD Type X will encompass all DataNode instances in a deployment that are associated with Type X HDDs attached to the nodes. This enables hat$S$ to not only capture the unique characteristics of heterogeneous storage devices but also distinguish between different storage tiers and utilize them accordingly. To achieve this, we modify the DataNode to also include a tier identifier as part of its characteristics specification. At the time of cluster configuration the administrator specifies the tiers for the DataNodes. We also modify the NameNode to use tier identifiers to group the DataNodes into their associated tiers. Each node can be part of multiple tiers depending on the devices that are attached to it. Moreover, a tier typically will have only one kind of device; but multiple kinds, such as HDDs that have only slightly different I/O characteristics, can also be associated with the same tier at the administrator's discretion. The number of DataNodes making up a tier vary based on the hardware composition of the cluster. In the example shown in Figure 4.1, Node 1 has three DataNodes belonging to three different tiers and Node 5 has only one DataNode belonging to Tier-3. Tier-1, Tier-2 and Tier-3 have 2, 3 and 5 DataNodes, respectively.

hat$S$ exploits the tier information to strategize when and where to place replicas of a block.

Table 4.1 hat$S$ APIs to enhance HDFS.

API	Arguments & Return Type	Description
boolean createFileTier(...)		**Creates a replica of a file in the specified tier.**
	String `filename`	Name of the file to be replicated.
	String `tier`	Tier in which the replica will be created.
	boolean `return_value`	Returns 0 on success, 1 on failure.
boolean deleteFileTier(...)		**Removes a replica of a file from a tier.**
	String `filename`	Name of the file whose replica will be deleted.
	String `tier`	Tier from which to remove the replica.
	boolean `return_value`	Returns 0 on success, 1 on failure.
boolean moveTier(...)		**Moves replicas of a file across tiers.**
	String `filename`	Name of the file to be moved.
	String `from_tier`	Source tier from which replica will be removed.
	String `to_tier`	Destination tier 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 setRepPolicy(...)		**Modifies the replication policy for a file.**
	String `filename`	Name of the file to be affected.
	String `policy`	Storage policy to use.
	int `number_of_replicas`	Number of replicas under the new policy.

We discuss several data placement policies in the subsequent section, however, hat$S$ maintains the invariant that a tier contains all blocks belonging to a file. This is to avoid dividing a file across a slow and a fast tier, where the slow tier devices will become the bottleneck and negate the benefits of the fast devices. Moreover, a tier can have more than one replica of a file, and a file can be replicated in multiple tiers as long as each tier contains a complete copy of the file. This provides for routing accesses to frequently used files to faster tiers, and relegating the infrequently used files to slower tiers. Note that, this approach still provides for replicating data within and across racks as in standard HDFS, but imposes the additional constraint of keeping a complete copy of a file in a given tier. One concern is that the whole copy invariant may be violated in case of node failures. To overcome this, we introduce a new monitoring daemon on the NameNode, which ensures that any re-replication is done in a tier-aware fashion.

4.1.2 hat$S$ Data Management APIs

In addition to the file system APIs provided by HDFS, hat$S$ provides new APIs specified in Table 4.1 to manage the tiered storage. The main functions of hat$S$ include associating DataNodes to appropriate administrator-specified tiers, and providing data access based on different policies. DataNodes are added to the tier during initialization only, and can not be modified at runtime. This is because the I/O and capacity characteristics that are considered in our tiers are specific to the devices used and do not change while the system is running.

Thus, the main runtime APIs allow the system to move files between tiers, create a replica of an already existing file in a specified tier, delete a file from a specified tier, and create new replicas based on specified replica management policy. We note that, similarly as in HDFS, all the APIs modify data placement in the granularity of a file and do not support block level modification.

4.1.3 Data Placement and Retrieval Policies

A challenge in hat$S$ is to determine when and where the data, i.e., a replica of a file, should be placed. This is a crucial design decision as naive replication can compromise performance and reduce the efficacy of our approach. Moreover, since hat$S$ like HDFS is a write-once read-many file system, provisioning for efficient data retrieval is also crucial for improving overall system I/O performance, and depends on the placement policy employed. In the following, we describe several data placement and retrieval policies that we consider.

4.1.3.1 Network-Aware Policy

The first policy that we consider is the default network-aware placement and retrieval used in HDFS. The placement policy distributes each data block across multiple racks to ensure fault tolerance against node and rack failure. However, network-aware data placement does not take into account the performance of underlying storage devices. Under this policy, the blocks are randomly distributed across DataNodes in a rack, so a file may be replicated across tiers such that the portion of the file stored in a tier will depend on the number of DataNodes in that tier. Similarly, 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. While this approach reduces network traffic, the nearest replica can be on a slower device. In contrast, a more distant but faster replica could have provided higher overall I/O and would have been a better alternative. Network-aware retrieval is oblivious to this information and hence cannot leverage such heterogeneity-based trade-offs. Thus, this policy is not a good match for hat$S$ as it crosses tier boundaries and will lead to performance imbalance when multiple types of devices are involved, e.g., SSDs and HDDs. Moreover, given that not all tiers are expected to be provisioned with the same amount of storage, and there will be more DataNodes in HDD tier (given the low $/GB) than tiers

containing expensive devices such as SSDs and ramdisks, this policy will direct most accesses to the slower devices even when faster devices are available in the system.

4.1.3.2 Tier-Aware Policy

The next policy we consider is tier-aware placement and retrieval, which takes into account the storage characteristics of the underlying storage devices and completely replicates a file in multiple tiers. For clusters having more than one storage tier, we replicate the file to up to three different tiers. For clusters with more than three tiers, we chose the first replica to be placed in a fast tier, the second in a slow tier, and the third in a randomly chosen tier with intermediate performance. Since the first replica is treated as a source for the second replica [203, 207], storing the first replica on a faster tier will also speedup the replication process. Tier-aware placement does not consider the underlying network infrastructure as such, and only ensures that a node stores a single replica even if the node has multiple DataNodes. This prevents data loss in case of node failure, as the replica can be re-created from other sources.

For data retrieval, the ideal would be to always access the data from the fastest tier. However, doing so in Hadoop will result in hotspots where some DataNodes are overloaded, and will affect the performance of the system. Moreover, given that the capacity and number of faster tiers is limited, retrieving data only from the fastest available tier will also entail higher cross-rack network traffic and related overheads. To this end, we associate a weight to each tier from which a block can be retrieved, and then employ a weighted random function to determine which DataNode to use for retrieval. The assigned weights of each tier are determined using storage characteristics, such as IOPS and capacity, of the DataNode. This approach is effective in distributing the requests to a file among multiple tiers and each tier will serve varying number of blocks. For example, an SSD with $70k$ IOPS will be able to serve at least $10\times$ more request than a HDD with $3.5k$ IOPS as we show later in the evaluation (Section 8.2).

While this policy takes into account tier characteristics, all the replicas of a block may be stored on one rack, and the data may be exposed to rack failures. Moreover, replication of blocks across racks is desirable for load balancing and providing better data locality for read operations. To avoid such data placement skewness, network characteristics have to be considered along with tier information, which we do in our next approach.

4.1.3.3 Hybrid Policy

Tier-aware data placement and retrieval policy improves the I/O performance by making replicas in specific tiers, while network-aware placement improves resilience by making replicas across racks. For improving I/O throughput, reducing cross-rack traffic to efficiently use the network, and fault tolerance, we need to have replicas across tiers as well as across racks. To this end, we design a hybrid network- and tier-aware data placement policy that works as follows. The first replica is placed with one of the DataNodes on the local node. The second replica is placed in a different rack than the one used for the first replica and also on a different storage tier than that of the first replica. The third replica is placed on a tier different from the other two replicas, but rack-local to either of the replicas. Moreover, the tier selection is done similarly as in the tier-aware placement policy. The key advantage of this policy is that it achieves the same replica distribution as that of standard HDFS, which is effective in ensuring high I/O with good resilience to failures, while also considering the heterogeneous storage characteristics.

Similarly, for retrieval we adjust the weights used in our tier-aware policy to also factor in network proximity and the cost of transferring a block over the network to achieve higher I/O throughput as well as to reduce cross-rack traffic.

While the hybrid policy has the same expected fault tolerance as in HDFS before a failure occurs, after a failure occurs special steps have to be taken by hat$S$ in replica regeneration to ensure that a new replica is stored on an appropriate tier in addition to being on an appropriate rack. Moreover, if a DataNode is overloaded with requests or low on capacity, replica creation or regeneration may not be possible on appropriate tiers. However, we then utilize the monitoring daemon to detect placement anomalies and move the data to appropriate tiers.

4.1.4 Discussion on hat$S$ Design

In this section, we discuss the impact of hat$S$ on other cluster components. First, the Hadoop job scheduler is network-aware and aims to schedule jobs on or near nodes that hold the needed data. Our hybrid approach preserves the network proximity, thus no change is required in the scheduler to avail the higher I/O rates offered by hat$S$.

Second, hat$S$ tries to utilize faster tier resources whenever possible. However, the number

of such devices is likely to be limited given their high cost. This would mean that the faster tiers may quickly become full, and the applications needing more data can no longer benefit from them. We remedy this by using the monitoring daemon along with replica movement APIs to flush the unused data from the faster to slower tiers.

Third, hat$S$ requires nodes to run multiple DataNode instances instead of just one as in standard HDFS. This can potentially increase the load on the node and affect performance. We argue that this additional overhead is distributed across all the nodes and is negligible because of the following reasons. (i) The different types of devices attached to a node is expected to be small. (ii) The total number of blocks stored on the node is similar as under HDFS and is independent of the number of DataNodes. The in-memory data structures at the NameNode depend on the number of data items and number of replicas, but not on the number of DataNodes. Since, we do not increase the number of replicas, we expect this factor to be the same as well, so hat$S$ is not expected to add any significant overhead to the NameNode. (iii) The overhead associated with accessing a block is also similar to HDFS, as hat$S$ modifies only the metadata space of these blocks.

Fourth, hat$S$ proposes to utilize SSDs in the Hadoop storage tier. There is a concern that such devices have limited erase cycles, and may affect the MTTF. We argue that incorporating SSDs in Hadoop is not unique to our approach, and other state-of-the-art works have also purported the same. Moreover, numerous SSD optimization approaches are available [208, 209] to remedy this. Thus, SSD endurance is orthogonal to our design; is useful even in when no SSDs are used but different kinds of HDDs are employed.

In summary, hat$S$ provides a variant of HDFS, which considers the characteristics of the underlying storage devices and network infrastructure for its data access policies, thus yielding improved I/O performance.

4.2 Implementation of hat$S$

We have implemented hat$S$ as described in Section 4.1. In total, we modified or added about 1800 lines of Java code in Hadoop 0.20.1 to add the features of tiering and heterogeneity awareness and to enable the APIs of Table 4.1.

4.2.1 Tier identification

We modify the `hadoop-daemon.sh` script to enable a Hadoop node to have multiple logical DataNodes, and to coalesce DataNodes with similar storage characteristics into respective tiers. We introduce a new parameter *dfs.tier.id* in the Hadoop configuration file (`hdfs-site.xml`), which the cluster administrator can use to identify the tiers for the different storage devices. Next, we modify HDFS's *DataNodeDescriptor* data structure to incorporate the tier information as an additional global characteristic of each DataNode. The extended descriptor can then be used by the HDFS's *DataNodeRegistration* process for registering the tier-based DataNode with the NameNode.

4.2.2 Data placement

To support data placement policies based on storage device characteristics, we modify the NameNode's *ReplicationTargetChooser* component to implement different data placement schemes. A list of nodes is chosen from the *NetworkTopology* structure that provides information about various racks and tiers in the cluster (*clusterMap*). To ensure that a DataNode is not used to store multiple replicas of the same block, we re-purpose the block-specific *excludenode* list by adding the already chosen DataNode as well as the other DataNodes on the same node to the list. This results in a node having only one copy of a block as desired.

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

4.2.3 Data retrieval

Data retrieval in HDFS uses weighted random approach to select a replica from the list of DataNodes that store the data. To support our different retrieval policies, we implemented weighted random methods in *NetworkTopology*. The weights can be re-adjusted for this selection based on the policy, i.e., network-aware, tier-aware or hybrid. For instance, network-aware scheme will assign weights to a DataNode based on its proximity to the client,

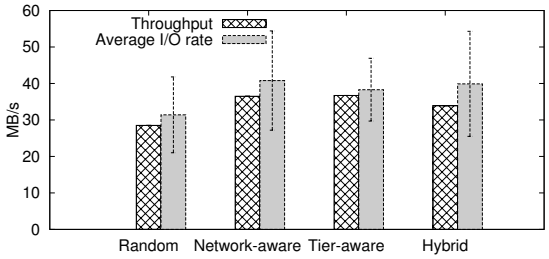


Figure 4.2 Overall write throughput and average I/O rate per map task in *TestDFSIO-Write* under the studied policies.

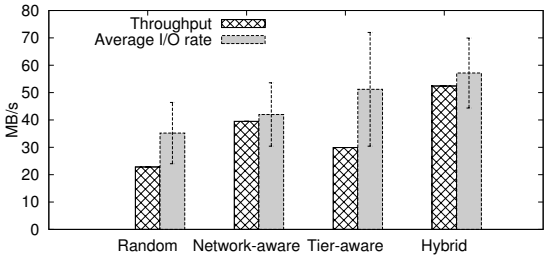


Figure 4.3 Overall read throughput and average I/O rate per map task in *TestDFSIO-Read* under the studied policies.

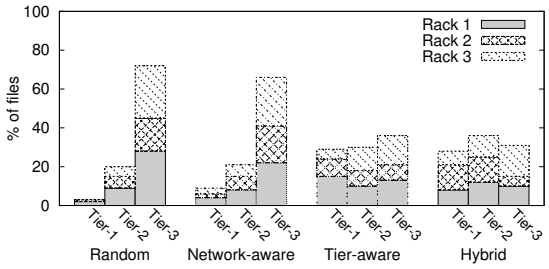


Figure 4.4 Data distribution in terms of tiers and racks under the studied data placement policies, normalized to the total data stored for each run.

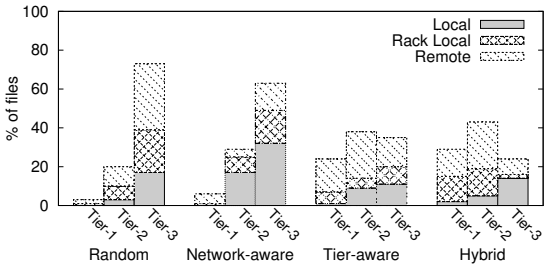


Figure 4.5 The breakdown of reads based on tier and network proximity. The y-axis is normalized to the total read accesses in each run.

tier-aware scheme will assign weights based on the characteristics of the storage device that the DataNode supports, and hybrid scheme will consider both the factors. In our current implementation, we have used fixed hard-wired values for the weights as our testbed characteristics are known to us a-priori, but in a real setup, the administrator can specify the weights in a configuration file.

4.3 Evaluation of hat$S$

In this section, we present the evaluation of hat$S$ using both a real deployment on a medium-scale cluster and simulations. We compare the effectiveness of different data placement and retrieval policies, and their impact on the I/O performance of Hadoop jobs. For comparison, we also consider a random data placement and retrieval policy, which is oblivious of both tier and network information.

Table 4.2 Specifications of different storage devices used in the HDFS test.

Device Type	Write BW MB/s	Read BW MB/s	IOPS	# of devices
PCIe SSD	245	533	70k	3
SATA SSD	139	191	25k	9
HDD	46	61	3.5k	27

4.3.1 Experimental Setup

Our testbed consists of a master node and 27 worker nodes configured in three racks of nine nodes each. The nodes have two 2.8 GHz quad-core Intel Xeon processors, 8 GB of RAM, and one SATA HDD. The HDDs are 500 GB 7200 RPM Seagate Barracuda ES.2 drives. In addition to HDDs, three of the worker nodes in each rack are provisioned with an Intel 520 series 128 GB SATA SSD and one worker node in each rack is provisioned with an additional OCZ RevoDrive series PCIe 128 GB SSD. Table 4.2 shows the performance specifications of these storage devices. In our setup, Tier-1, Tier-2, and Tier-3 contain all the DataNodes that are equipped with the PCIe SSDs, the SATA SSDs, and the HDDs, respectively. Moreover, a node is associated with at most two tiers. The nodes are connected using both a dedicated 1 Gbps Ethernet switch as well as a dedicated 10 Gbps InfiniBand switch. We use InfiniBand as our default interconnect, using the slower connection only where specified in the following discussion. Each worker node is configured with six map slots and two reduce slots so as to use all of the available cores on the node. The considered benchmarks are mostly map intensive, so there are more map slots than reduce slots.

The master node runs both the Hadoop JobTracker and NameNode for all the experiments, and all the worker nodes contribute to both TaskTracker and DataNode. Worker nodes with more than one type of storage devices have multiple DataNodes, thus our testbed has 39 DataNodes co-existing with 27 TaskTrackers. As the focus of our experiments is to study the impact of HDFS I/O operations, the intermediate shuffle data is stored on the HDDs local to the TaskTracker. The replication factor is fixed at the default three, and the block size used is 64 MB.

4.3.2 Performance Under Different Policies

We analyze the read and write performance of hat$S$ under different data placement and retrieval policies using the HDFS benchmark *TestDFSIO*. Each worker node writes a 1024 MB file (16 blocks) during the write test and reads a file of the same size during the read test.

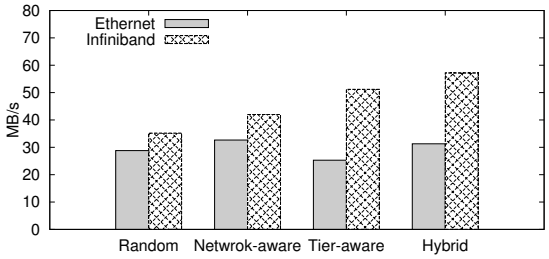


Figure 4.6 Average I/O rate observed for InfiniBand and 1 Gbps Ethernet connections under studied data management policies.

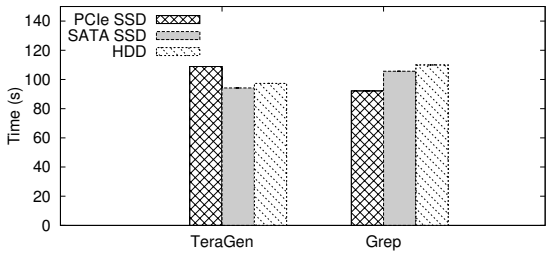


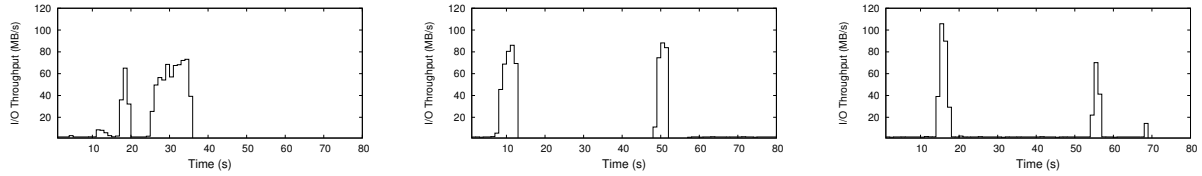
Figure 4.7 Comparison of execution time observed for the different storage devices.

Figure 4.2 shows the results for the write accesses. We measure the overall I/O throughput for each of the map tasks and calculate the average I/O rate across all map tasks. We observe that the network-aware and hybrid policies behave similarly. This is because the write operation succeeds after it writes to the OS buffer cache and does not wait for the data to be synced to the storage device. We see a reduction in the throughput and average I/O rate for the tier-aware and random policies, which is expected as they do not consider network proximity and associated overhead.

Figure 4.3 shows the *TestDFSIO* results for the read test. As the hybrid policy considers network proximity and tier information, it offers significantly higher I/O rates than the other studied policies. Similarly as in the write test, the network-aware and the tier-aware policies perform better than the random policy. An interesting observation here is that the network-aware policy has a higher throughput than the tier-aware policy, whereas the average I/O rate of the tier-aware policy is better than that of the network-aware policy. The tier-aware policy is network oblivious, thus the probability of using a local fast tier for an access is similar to that of using a remote slow tier, which is seen as a high standard deviation in the average I/O rates in the Figure. Moreover, the tier-aware policy aggressively tries to utilize the storage devices in the fast tier without considering the network constraints. This sometimes results in network contention, causing the I/O rate to be quite low for some map tasks. We also observe that the hybrid policy offers 32.6% better I/O throughput and 36% better average I/O rate compared to that of the default network-aware policy.

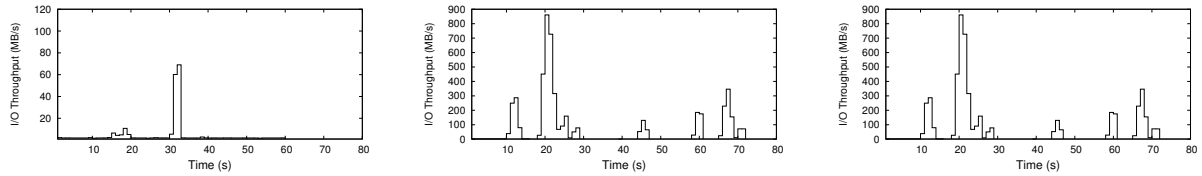
4.3.3 Impact of Placement Policy

In our next experiment, we used *TeraGen* to generate a 27 GB file consisting of 432 blocks. By default, *TeraGen* uses only two mappers to generate all of its data, so each TaskTracker



(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Figure 4.8 Disk usage under the network-aware policy.



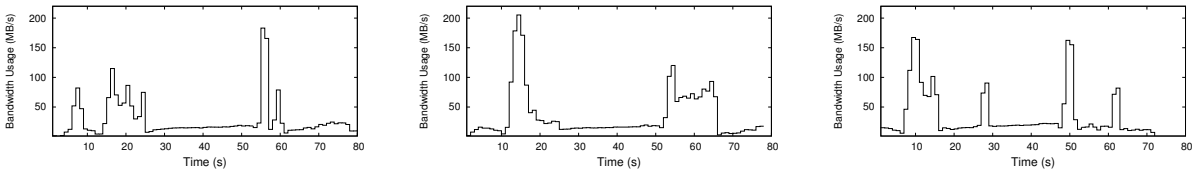
(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Figure 4.9 Disk usage under the hybrid policy.

generated 13.5 GB. In the network-aware and hybrid placements, while storing the local replica of the data, all the blocks of the 13.5 GB file will be skewed to one DataNode (corresponding to the TaskTracker generating the data). To avoid this, *TeraGen* uses one mapper per TaskTracker, with each mapper generating a 1024 MB (16 blocks) file.

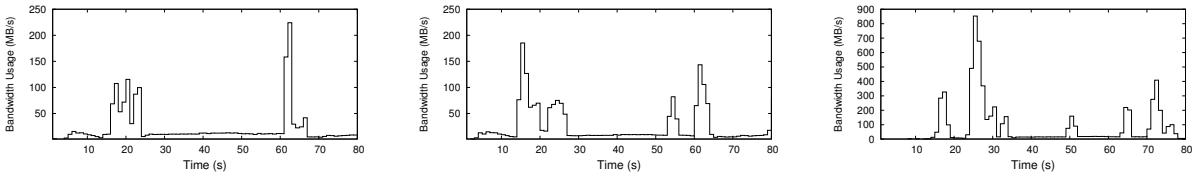
Figure 4.4 shows the distribution of files across racks and tiers. To obtain this information, we parse and analyze HDFS's block map that stores information about all the blocks associated with a DataNode.

For the network-aware policy, we find that more replicas of a file are placed in Tier-3 than in Tier-1 and Tier-2. This is because there are a fewer number of Tier-1 and Tier-2 DataNodes in comparison to Tier-3. In this policy, the distribution of replicas across the tier is directly proportional to the number of DataNodes contained in the tier. The results for the tier-aware and hybrid policies reveal replication of a block across all tiers. Since all the racks have the same number of nodes, even the network oblivious policies – random and tier-aware – have equal number of replicas across racks. The difference between these and the network-aware policy is that, in the later, each block is replicated across multiple racks to achieve resilience against rack failures. For the case of network oblivious policies, we see that 12% of the blocks are replicated only within one rack and will be exposed to data loss in case of a rack failure.



(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Figure 4.10 Network usage under the network-aware policy.



(a) DataNode with HDD. (b) DataNode with SATA SSD. (c) DataNode with PCIe SSD.

Figure 4.11 Network usage under the hybrid policy.

4.3.4 Impact of Retrieval Policy

In the next set of experiments, we study the role of the retrieval policies of hat$S$. We used *Grep* to read the data generated by *TeraGen* using six mappers per TaskTracker. Figure 4.5 shows the results. We observe that the random and the network-aware retrieval policies do not read a large number of files from the faster Tier-1 or Tier-2. This is mainly due to the fast tiers having a fewer number of blocks. Moreover, the probability that an access to a replica will be sent to a specific tier depends on the number of DataNodes in that tier, thus a tier with fewer blocks have fewer accesses. We find that the network-aware policy has 22% and 33% less remote requests than the random and tier-aware policies, respectively.

As expected, the tier-aware and hybrid policies access more requests from the fast tiers compared to the slow tiers. We observe that the hybrid policy results in 4× more accesses to the Tier-1 than the network-aware policy, and only 13% more remote accesses than the network-aware policy. Further examination reveals that the hybrid policy results in 30% more accesses to Tier-1 and Tier-2, though at the cost of 15% increase in non-node-local (rack-local and remote) accesses. This trade-off between tier and network awareness offers an effective control knob that can be modified based on the infrastructure provisioning of a cluster to maximize performance.

4.3.5 Impact of Network Speed on hat*S* Performance

In the next set of experiments, we compare the average I/O rate of the studied data management policies under our two testbed interconnects: 1 Gbps Ethernet and InfiniBand. In Figure 4.6, we see that the average I/O rate under InfiniBand is better than that achieved under the 1 Gbps Ethernet. While expected, this result serves as a sanity check that our enhancements do not have unintended side-effects. Moreover, in the 1 Gbps Ethernet setup, we find that the network-aware policy performs better than other policies. We observe no additional advantages of the tier-aware policies with 1 Gbps Ethernet. In case of the random policy, the performance is better than the tier-aware policy. This is because of the network contention for the fast tier DataNodes as more requests are routed to them. From this test, we observe that better network provisioning is necessary to avail the benefits of hat*S*. However, this is not a limitation, as better interconnects are typical in enterprise data center deployments.

4.3.6 Network and Disk Utilization in hat*S* DataNodes

Next, we compare the network and disk usage of hat*S* for the read operation under two policies: network-aware and hybrid. For this purpose, we repeated the test described in section 4.3.4 and used SAR [210] to collect detailed disk and network usage statistics for the DataNodes. Figures 4.8, 4.9, 4.10, and 4.11 show the behavior of one DataNode in each tier; similar patterns were observed for other DataNodes in the respective tiers.

As shown in Figure 4.8, under the network-aware policy, the HDD utilization is 40% higher than the combined utilization of the SATA SSD and the PCIe SSD. This highlights the disadvantage that the network-aware policy does not effectively utilize the expensive high performance storage devices. In contrast, Figure 4.9 shows the disk usage statistics under the hybrid policy. In this case, the number of requests to the DataNodes contained in Tier-3 are minimized. The SATA and PCIe SSDs together service 36% more read accesses than HDDs, and effectively utilize their high I/O bandwidth. Under hybrid policy, the utilization of PCIe SSDs has increased by 91%, and its maximum I/O throughput is 10× of that achieved under the network-aware policy. This shows the advantages of hat*S* over standard HDFS in better managing the heterogeneous storage devices.

Similarly, Figure 4.10 and Figure 4.11 show the network throughput of DataNodes belonging to different tiers under the network-aware and hybrid policies. The network utilization

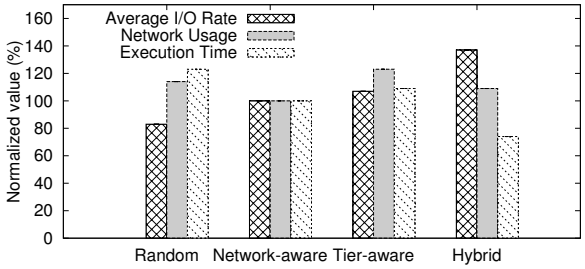


Figure 4.12 Comparison of the average I/O rate, network usage, and execution time under the studied policies normalized to case of the network-aware policy.

is observed to be almost the same for DataNodes with the HDD and SATA SSD, whereas for DataNodes with the PCIe SSD the maximum bandwidth throughput is very high. This is because, under hybrid policy, three PCIe SSDs serve 28% more requests than under the network-aware policy, resulting in an increase in remote accesses.

4.3.7 Impact of Storage Characteristics on Hadoop Performance

In out next experiment, we study the impact of different storage devices on Hadoop. For this test, we provision HDFS to service all the requests from only one type of device for its storage. Our testbed contains only three PCIe SSDs but 27 HDDs, so as to ensure fairness and avoid performance bottleneck due to network contention for the SSD DataNodes, we reduce the number of worker nodes for this experiment to five nodes per rack. Each rack contains one PCIe SSD, three SATA SSDs and five HDDs. Figure 4.7 compares the execution time of *TeraGen* and *Grep* for three cases: HDFS with three PCIe SSDs, with six SATA SSDs, and 15 HDDs. *TeraGen*, which is a write-intensive application, performs better with HDDs and SATA SSDs than with PCIe SSDs. This is similar to our observation in section 4.3.2 for *TestDFSIO-write*.

After each benchmark, we clear the contents of the DataNodes' buffer caches to prevent cross-benchmark pollution. For *Grep*, which involves a significant amount of read operations, we find that even though there is a small number of the PCIe SSDs, they perform significantly better than the SATA SSDs and HDDs. The three PCIe SSDs perform 20% faster than the 15 disks. We repeated the experiment with 21 worker nodes and found that the read performance of the three PCIe SSDs was similar to that of 21 HDDs.

4.3.8 Simulation-Based Experiments

For our next set of experiments, we developed an accurate simulator for hat$S$ to observe the behavior of the considered data management policies on a large cluster setup. Our fine-grained simulator takes into account details such as the effect of intermediate shuffle data, network and storage infrastructure, and application I/O patterns. We simulated a 500-node cluster, with each node equipped with six 1 TB disks and one 256 GB PCIe SSD. These nodes are interconnected using two 10 Gbps InfiniBand links. We use the publicly available synthetic Facebook production traces [211] for driving the simulation. We replay the traces using HiBench [212] applications to process 70 TB of input data, generate 17 GB of intermediate shuffle data and 13 GB of output data spanning over three weeks. To make room for newly generated data, we use Least recently used (LRU) policy to evict the data.

Figure 4.12 compares the average I/O rate (the higher the better), network usage and trace execution time for the studied policies normalized to the case of the network-aware policy. We observe that the hybrid policy yields a 37% higher I/O rate as compared to the network-aware policy, and at the cost of 9% increase in the network usage (the lower the better). The tier-aware policy results in the highest network usage, i.e., 23% more than the network-aware policy. Data placement and retrieval behavior observed in the simulations is similar to that observed for the real testbed experiments. We see that overall SSD usage was improved by 68% with over 52% of the data accessed from the PCIe SSD. Finally, we also study the execution time (the lower the better) of the benchmarks under different policies. Our hybrid policy offers the best execution time, which is 26% better than the extant network-aware policy.

In summary, our evaluation of hat$S$ reveals that it offers a viable solution to enhancing HDFS to incorporate heterogeneous storage, and does so efficiently. Our hybrid data management policy captures both tier awareness and network awareness to offer higher I/O rates and reduced execution time. These features are key to sustaining Hadoop for emerging architectures and applications.

4.4 Chapter Summary

In this chapter, we presented a dynamic data management scheme for Hadoop for achieving higher throughput and lower storage cost. We observe that managing all Hadoop data in a

uniform manner results in increased storage overhead or reduced read throughput. For popular files, default replication is insufficient and leads to decreased throughput. For unpopular files, default replication results in storage inefficiency. We proposed AptStore, a system that exploits the heterogeneity in access patterns to achieve overall reduction in storage cost and increase in read throughput. We identify various factors that affect the throughput of the system and propose PPA to predict the popularity associated with each file, and use the information to adjust the replication and data placement strategy of the files. Using extensive simulations and a real deployment, we demonstrated that AptStore data management scheme increases the read throughput by 23.7%, reduces overall storage utilization by 43.4%, and results in speeding up the studied jobs by as much as 21.3%.

Chapter 5

Dynamic Storage Management for Hadoop

Hadoop Distributed File System (HDFS) [90] provides a robust storage for managing massive amounts of data in a scalable manner by aggregating the direct attached storage (DAS) of Hadoop cluster nodes [10]. The off-the-shelf machines that make up typical Hadoop clusters and the scale of the system imply that failures are the norm. To prevent data loss, HDFS relies on replication [90]. Replication also increases the read throughput, not only because it reduces access contentions that can arise when accessing popular data, but also by increasing the probability of finding the data on a local DAS.

While DAS with replication offers significant throughput benefits in Hadoop, the default three replicas also incur a 200% storage overhead. Not only does this overhead add to the direct cost of the storage, it has indirect maintenance costs of energy consumption and administration, which can be significant [213]. Another limitation of the DAS-based Hadoop architecture is that storage capacity is tightly coupled with compute capacity; to add more storage, more compute nodes need to be added. Thus increasing storage capacity in standard DAS-based Hadoop also incurs the cost for compute components, which may be unnecessary for typically I/O-bound Hadoop applications. Adding a whole node for just using the extra storage exacerbates energy efficiency as well, as typically, storage accounts for only a fraction of a Hadoop node's energy consumption [214].

To this end, Network Attached Storage (NAS) can offer an alternate storage solution for Hadoop, especially enterprise NAS is attractive due to its lower failure rates. To add to this, the per GB storage cost in enterprise storage solutions [92] is only a fraction of that in a commodity Hadoop node DAS. However, the challenge is that naively adding NAS to Hadoop clusters may entail a large number of data accesses over the network, resulting in reduced I/O throughput. A promising trend observed in recent analysis is that there is

significant heterogeneity in I/O access patterns. GreenHDFS [113] observed a news server like access pattern in HDFS audit logs from Yahoo, where recent data is accessed more than the old data and more than 60% of used capacity remains untouched for at least one month (period of the analysis). Scarlett [43] analyzed job history logs from Bing production clusters and observed that 12% of the most popular files are accessed over ten times more than the bottom third of the data.

We design a tiered storage system, AptStore, with two tiers designed to better match the heterogeneous Hadoop I/O access patterns. The tiers include: Primary storage — DAS in Hadoop node for files that require high throughput; and Secondary Storage — NAS for unpopular files and files with lower Service Level Objectives (SLO). AptStore analyzes the I/O access patterns and suggests data placement policies across the tiers to increase the performance and efficiency of the storage system. Our system optimizes for read throughput as typically MapReduce workloads exhibit write-once read-many characteristics [90]. To achieve this, we predict the popularity of each file, and then retain the popular files in primary storage and move unpopular files to secondary storage. We also adjust the replication factor of files in primary storage based on their popularity. The replication factor for files in the secondary storage is set to 1, and other means such as RAID are employed to achieve fault tolerance. We have realized AptStore as an extension to the Unified Storage System (USS) [91, 92], a federated file system for Hadoop, which allows transparent movement and management of data across different file systems.

5.1 Factors Affecting Hadoop Storage Performance

In the following, we discuss the key factors that impact the performance and efficiency of the storage system in Hadoop.

5.1.1 Understanding Read Throughput

There are two key factors that affect read throughput in HDFS: data locality and number of concurrent accesses. Local accesses result when a job and its associated data reside on the same node, thus reducing the number of remote I/O requests and yielding higher throughput. On the other hand, many concurrent accesses to the same file increase contention, thus decreasing read throughput.

5.1.1.1 Locality

HDFS divides the data into equal sized blocks and distributes data to multiple nodes, which distributes the read request throughout the cluster, thereby achieving better aggregate throughput. Block size is an important tunable parameter in the system. Bigger block sizes decrease the overall number of blocks per file and hence the number of nodes that hold data. This decreases probability of the job scheduler assigning tasks that are local to the data. However, decreasing the block size too much is also undesirable as it can result in memory contention in the *NameNode*. With constant block size, file size has a direct effect on the distribution of the data; larger files are distributed throughout the cluster, while the smaller files are restricted to a small set of nodes.

Replication also affects locality and in turn the read throughput in a Hadoop cluster [90]. Higher replication factor increases the distribution of the data in the cluster, thereby increasing the probability of *JobTracker* finding a local or rack local slot for a task. This results in reduced network and disk contention, particularly when multiple jobs access the same data concurrently. Thus, a small file with more replicas can have the same distribution as that of a larger file.

5.1.1.2 Concurrent Access

Concurrent jobs accessing a single block, or blocks from a single machine, not only affect the disk bandwidth available per access, but also the network bandwidth. In Hadoop clusters, concurrent access to popular data are common [43]. In such cases, when the number of tasks accessing the data exceeds the number of replicas, read throughput of the tasks is affected because of slot contention and hardware resource contention [102].

Large number of concurrent tasks reading data from a node decrease the probability of scheduling a task local to the data, and as a result read throughput includes network overhead. Since concurrent requests share the disk bandwidth and network bandwidth, the number of concurrent accesses is inversely proportional to the read throughput. The scenario is typical in production clusters, especially on machines storing popular data. Scarlett's [43] analysis of Bing production cluster indicates that more than 50% of read requests were directed to less than 17% of the cluster. The decrease in read throughput because of increased concurrent accesses can be reduced by increasing the replication of the file and distributing the requests across the cluster.

5.1.2 Fault Tolerance

Since Hadoop clusters are built using commodity machines, the hardware failure rate is non-negligible. The typical Mean Time Between Failures (MTBF) is 3 years [184], so for a thousand node Hadoop cluster, the probability of failure of a single machine in the cluster is close to one. Data loss prevention using RAID is not a feasible solution because equipping each Hadoop node with a RAID controller is expensive and software RAID on unreliable machines incurs high performance overhead. Since availability is proportional to MTBF, reducing the replication factor to 1 and using parity to prevent data loss might result in reduced availability. Moreover the low reliability of the hardware implies periodic loss of data resulting in reconstruction from the parity. Such generation and reconstruction will adversely affect the performance of the in-progress Hadoop jobs.

In contrast, a more feasible RAID based solution can be used by employing consolidated NAS if high I/O throughput is not a concern. Enterprise storage solutions typically utilize RAID and have lower MTBF [215]. These devices ensure the same reliability of data with significantly less storage overhead. Moreover these devices are self managed and reconstruction of parity would not affect the in-progress Hadoop jobs, unless the jobs are trying to access the files under recreation. Thus, incorporating NAS into Hadoop architecture is promising and can support low-cost fault-tolerant storage.

5.1.3 Storage Cost

The use of replication increases the capacity needed to be provisioned and thereby exacerbates the cost associated with the storage. While DAS offers better performance at higher cost, enterprise filers offer degraded performance at lower cost. Thus, it would be beneficial to utilize the different kinds of storage in realizing an efficient Hadoop storage architecture, provided the performance requirements for the data items can be determined or predicted.

Typically, a Hadoop node with a maximum of 24 TB of data storage uses up to 200 W [216] at idle state. The energy cost of adding a Hadoop node for storage scalability would result in 8.33 W/TB. Along with the 200% storage overhead the energy cost of storing data in DAS is 25 W/TB. This is very high when compared to the 0.78 W/TB in enterprise storage solutions [217]. Thus, from the energy consumption perspective, use of NAS in Hadoop is very favorable.

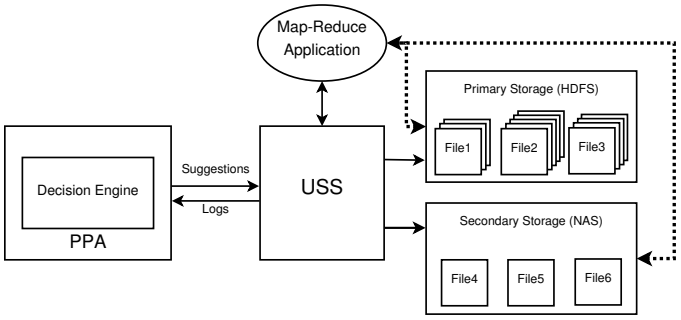


Figure 5.1 AptStore architecture overview.

5.2 Design of AptStore

In this section, we describe the design of AptStore, including its decision engine, the Popularity Prediction Algorithm (PPA), as well as how AptStore is realized within an available enterprise NAS implementation.

5.2.1 AptStore

AptStore is designed as an extension to the Unified Storage System (USS) [91, 92] [1], a federated file system for Hadoop, which allows tiered storage across different file systems. As access rate and number of accesses varies for each file, AptStore improves the overall read throughput and storage efficiency of the system by designing access-pattern-based data placement and replication. Figure 5.1 illustrates the components of AptStore and their interactions. The PPA periodically analyzes the usage patterns of the file system and the Decision Engine (DE) suggests appropriate data placement strategy.

In designing AptStore, we make the following four design choices. First, we consider replication at file granularity because Hadoop jobs access files as a whole [43]. Making replication decisions at the block level is unnecessary, as the read cost of a file will be dependent on the block with the lowest replication factor. Second, we assume that typical production clusters are heavily used and have very large working sets. Thus, any effects of file system level caching is negligible, and the majority of reads are serviced from disk. Third, all the files in the system are assumed to have the same block size, which is standard in Hadoop deployments. Finally, AptStore is designed for the extant Hadoop deployments where all nodes contribute storage and computation.

[1]Note that techniques developed in AptStore are not USS-specific and can be easily implemented and integrated with other NAS solutions.

5.2.2 Unified Storage System

The Hadoop framework works best when used in conjunction with HDFS. The Hadoop command line (`FsShell`) and file system API are supported only by HDFS and its variants [100]. Jobs involving data from other sources are processed by first loading the needed data into HDFS, typically by using tools such as `cron`, `scp`, and `distcp`. Another solution to multi-source data access is to use viewFS [218] or add data sources directly using Uniform resource identifiers (URI). However, adding enterprise storage devices through these approaches lead to load imbalance and decreased read throughput, mainly because the devices would likely not be available centrally/equidistant across the cluster.

To address such issues, USS implements a federated file system that provides a unified view using a single namespace that encompass a multitude of data sources. USS supports transparent, zero-copy access of data from various data sources. It also maintains a mapping of all HDFS files to their actual locations in the respective file systems. We leverage and extend this feature in AptStore to transparently move data between primary and secondary storage as needed.

5.2.3 Popularity Prediction Algorithm

We design a Popularity Prediction Algorithm (PPA) using file access information to determine when and where to store the files. At every RT, the PPA analyzes the access pattern for each file and predicts a expected popularity value for it for the next RT. The popularity value $P_{i+1}(f)$ of a file f varies with each access $i+1$ to the file. $P_{i+1}(f)$ is defined as:

$$P_{i+1}(f) = P_i(f) + \frac{c}{a(f) * l * b(f) * P_i(f)}, \tag{5.1}$$

where c is the popularity constant, $a(f)$ is a function of the access interval of file f, l is the load in the cluster and $b(f)$ is a function of number of blocks in the file f. Observe that we designed our popularity measure to recursively depend on the file popularity during the previous time interval. This causes the number of replicas to remain stable even in the presence of a bursty access patterns between successive intervals, yet adapt to changes that are longer lasting. Additionally, this allows the system to adapt effectively to access patterns of periodic jobs or ones that are scheduled at intervals wider than RT.

Equation 5.1 also ensures that the popularity of file f increases not only with the number

of accesses but also if it is accessed concurrently by many clients. The access frequency is inversely proportional to the time between the previous access, i, and the current access, $i + 1$. During any RT, files with the same number of accesses may have different access frequencies. For example, a file can have one access every five minutes for a total of 12 accesses in an hour, whereas another file may have 12 concurrent (non-repeating) accesses. Reads with higher access frequency require more replicas of the accessed files than those exhibiting lower accesses frequencies, even if the total number of reads within a RT are the same. This is because frequent reads cause contention both at the disk and network, resulting in degraded read throughput.

The required replication factor also depends on the cluster load, l, computed using the overall popularity of all files in the system. Many concurrent requests for multiple files can compound and result in an increase in contention for both the disk and the network bandwidth. Although increasing the cluster infrastructure to handle a higher load is one possible solution, it is not always feasible. Our solution is to aggressively replicate popular data, because it would better distribute the requests across the cluster and increase the probability of accessing the data locally. Conversely, we reduce the number of replicas for unpopular data.

As we assume a constant block size, larger files have more blocks. Consequently, larger files are better distributed throughout the cluster, so they require fewer number of replicas than smaller files. To capture this aspect, we update the popularity of the file after each access by an increment that is inversely proportional to the file size.

During the creation of a file, the popularity of the file $P_1(f)$ is initialized to average file popularity observed in the system, $AVG(P)$. Initialization based on observations such as the type of jobs accessing the file or popularity of other data created by the same user are also promising, but we leave that for future work. Similarly, whenever a file is deleted, it will result in popularity of other files being modified when the values are updated at the end of RT. When a popular file is deleted, the popularity of other files in the system increases. Conversely, when an unpopular file is deleted, the popularity of other files decreases. We do fix the minimum, P_{Min}, and maximum, P_{Max}, threshold on the popularity of a file to make sure that there are bounds on the number of file replicas in the system. The minimum threshold ensures data reliability and compliance with system SLAs, while maximum threshold captures space constraints in primary storage.

After the accesses of all files in the reference time RT are processed the popularity value

Input : USS file System Audit Logs
Output: Predicted popularity $P_{predicted}$.
F is the set of files in the file system;
foreach *access $i+1$ to the file $f \in F$ in RT* **do**
  if *i==0* **then**
    $P_{i+1}(f) \leftarrow AVG(P)$;
  end
  else
    $P_{i+1}(f) \leftarrow P_i(f) + \frac{c}{a(f)*l*b(f)*P_i(f)}$;
  end
  if $P_i(f) < P_{Min}P$ **then** $P_i(f) \leftarrow P_{Min}$;
  ;
  else if $P_i(f) > P_{Max}$ **then** $P_i(f) \leftarrow P_{Max}$;
  ;
  $IP = IP + P_{i+1}(f) - P_i(f)$;
end
foreach *deletion of the file f in F* **do**
  $IP = IP + AVG(P) - P(f)$;
end
$MIP \leftarrow \frac{IP}{size(F)}$;
foreach *file f in F* **do**
  $P_i(f) \leftarrow P_i(f) - \frac{MIP}{s}$;
  where i is the most recent access to the file f.
  $P_{predicted}(f) \leftarrow P_i(f) + (P(f) - P_i(f))$;
  $P(f) \leftarrow P_i(f)$;
end

Algorithm 1: Popularity Prediction Algorithm.

$P_i(f)$ of a file f for the most recent access i is modified as follows:

$$P_i(f) = P_i(f) - \frac{MIP}{s}, \tag{5.2}$$

where MIP is the mean increase in the popularity of the file f during reference time RT, $AVG(P)$ is the average popularity of all the files in the cluster, s is the scalability constant. Equation 5.2 ensures that the popularity of the file $P(f)$ does not grow arbitrarily. The mean increase in popularity is a fraction of increase in popularity, IP during RT over F, the set of all files in the system. The scalability constant s, is used to contract or expand the amount of data stored in primary storage. For a value of s greater than one, more data is pushed to primary, while a positive value of s, less than one, creates more space in the primary storage.

The choice of RT is critical. A very large RT can miss opportunities to change the file replication factor to adapt to a change in the access pattern as the workload varies. However, setting RT too small can result in excessive thrashing as PPA state is rapidly updated. The appropriate value of RT depends on the usage pattern of the cluster and the cluster infrastructure. Previous work [43] suggest an RT between 12 and 24 hours is sufficient to capture varying patterns in the workload, while minimizing overheads associated with managing extra replicas.

Finally, AptStore adopts a proactive prediction scheme. The predicted popularity, $P_{predicted}(f)$, for the the next RT is computed using a linear extrapolation. We assume that the rate of change of popularity for the next RT will be same as the rate of change of the current RT. By choosing an appropriate size of RT, the accuracy of the predicted popularity can be made high.

5.2.4 AptStore's Decision Engine

At every reference time RT, the decision system suggests the replication and data placement strategy for a file f based on $P_{predicted}(f)$, the predicted popularity of the file for the next RT . Files with higher popularity are replicated based on the function $P_{predicted}(f)$ and are placed in the primary storage. Files with lower popularity are moved to secondary storage and the replication factor is reduced to 1. Files with average popularity are maintained in the primary storage with default replication levels.

The system also considers cron jobs or jobs that are scheduled for later execution and predicts the appropriate storage strategy, thereby improving the read throughput. Finally, the system considers the SLA requirement irrespective of the popularity. For example, an unpopular files, although accessed rarely, may have significant SLA restriction, so it may be always replicated and stored in the primary storage.

5.2.5 Replication and Inter-tier Data Movement

Hadoop performance is sensitive to network bandwidth. Replication or data movement across tiers during such network intensive phase may adversely affect the performance of the Hadoop jobs in progress. To balance the bandwidth consumption, HDFS employs multi-location replication [43], where the data to be replicated are read from multiple sources

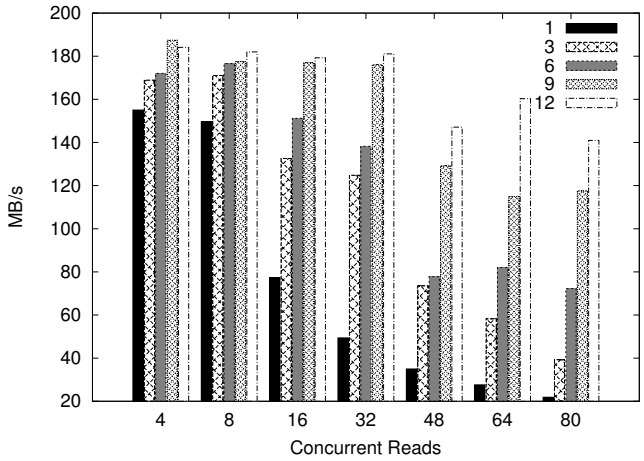


Figure 5.2 Read Bandwidth with increasing replication factor and with increasing number of concurrent reads.

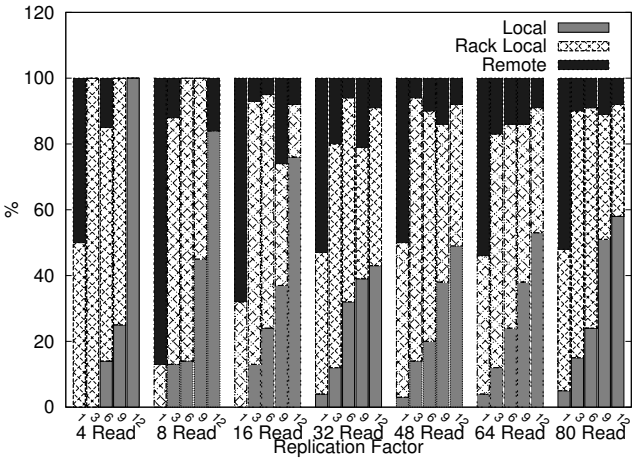


Figure 5.3 Number of Local vs. Rack Local vs. Remote with increasing replication factor and with increasing number of concurrent reads.

thereby spreading the replication traffic across multiple nodes. File movement between primary and secondary storage as well as change in replication factor is realized by a low priority background process.

System performance may further increase with a rack-level dedicated link from primary storage to secondary storage. Furthermore, significant work is done to improve the network utilization and storage utilization in Hadoop by compressing the data [219], which can be leveraged. For replica deletion, lazy deletion [43] of data, i.e., waiting for it to be overwritten by another block, may significantly reduce the cost of deletion.

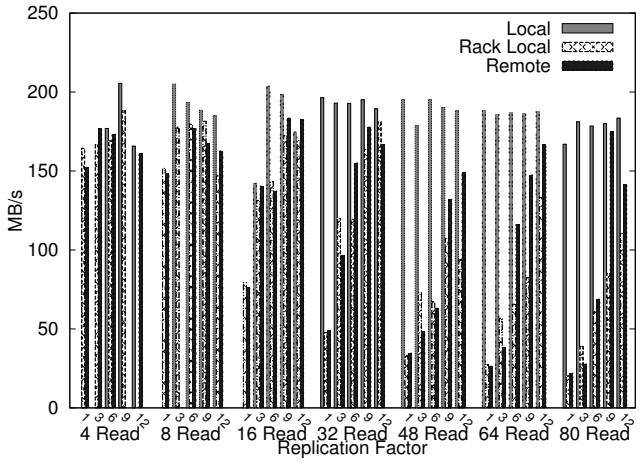


Figure 5.4 Bandwidth of Local vs. Rack Local vs. Remote with increasing replication factor and with increasing number of concurrent reads.

5.2.6 AptStore Computation Overhead

AptStore requires calculation of each file's popularity at the end of every RT. This computational overhead is negligible, because popularity is computed by linearly processing the file system audit logs, which the NameNode already generates. Hence, from the point of view of the TaskTrackers, our system produces negligible overhead for typical cluster sizes. For small clusters, PPA algorithm can run on the same node as the NameNode. However, for a very large cluster where popularity computation may incur some overhead, the algorithm can be offloaded to a separate machine.

The design of AptStore aims to monitor file I/O and utilize the PPA to determine the popularity of individual files. Our system extends USS to use the popularity information to move the files between primary and secondary storage tiers, thus providing high replication and high throughput for popular data, and low-overhead high-volume cheaper storage for unpopular data. Moreover, the use of PPA ensures that the AptStore is able to adapt to the changing characteristics of the Hadoop workloads.

5.3 Evaluation of AptStore

In this section, we present a detailed evaluation of factors that affect read performance in Hadoop and evaluate the performance of AptStore.

5.3.1 Experimental Setup

We use a 28 node cluster for our experiments. The master nodes have 3.33 GHz 2×Intel Xeon X5680 6-core CPUs with hyper-threading, 64 GB of RAM, and up to 3 SATA disks. The worker nodes have 3.33 GHz 2×Intel Xeon X5680 6-core CPUs with hyper-threading, 48 GB of RAM, and 12∗600, 15 K RPM disks. The master and workers are equipped with four and two network ports, respectively, and are interconnected using 10 Gbps link. There are two master nodes, one running a dedicated *NameNode* and the other the *JobTracker* and the *SecondaryNameNode*. Moreover, there are 26 worker nodes, each with an instance of *TaskTracker* and *DataNode*. The default HDFS block size is 512 MB and the version of Hadoop we employ is GPHD 1.2.

5.3.2 Impact of Design Parameters

In the first set of experiments, we analyze how various factors impact read throughput of Hadoop, and quantify the impact under different test conditions. To compute the read cost and eliminate any computation cost, we run a map-only job that reads a block of data. We execute this job with varying number of concurrent reads and on data sets of same size with varying replication factor. To minimize the effect of caching, we flush the file system caches on all disks contributing to HDFS between test runs.

5.3.3 Read Bandwidth Comparison

We observe the read bandwidth by varying the replication factor and the number of concurrent reads. Figure 5.2 shows the results. We find that 3 replicas provide sufficient for workloads with up to 8 concurrent reads and adding an more replicas produces marginal improvement in read throughput. As we increase the number of concurrent reads and thus the contention, more replicas are required to sustain the read bandwidth. For a workload with 80 concurrent accesses, any replication below 9 suffers significant loss in throughput. Conversely, in a workload with up to 32 concurrent accesses, increasing the number of replicas beyond 9 produces no performance benefit, thus wasting storage space used by the extra replicas. The results show that using a uniform replication factor is problematic in terms of both meeting throughput demands for popular files, and conserving space for unpopular files.

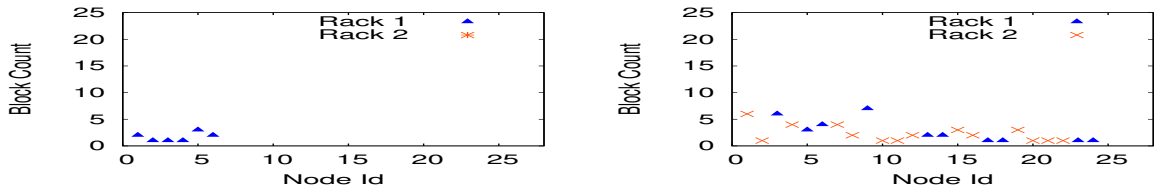
5.3.4 Impact of Access Locality

Next, we repeat the previous experiment but study the percentage of read requests that are served locally versus remotely. The results are shown in Figure 5.3. The percentage of local accesses increases with the replication factor. While the dataset with a replication factor of 12 achieves more than 40% of local accesses in all of the concurrent reads, using a replication factor of 3 achieves only a maximum of 14% local accesses.

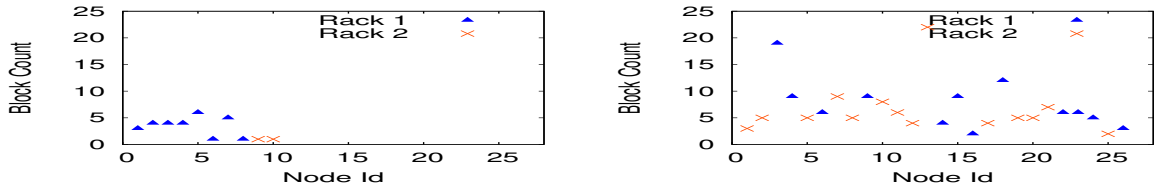
Bandwidth of local, rack local, and remote access are compared in Figure 5.4. We find that the local access use similar bandwidth across varying concurrent number of reads. With increasing concurrent accesses, there is a difference between bandwidth for the rack local and remote rack accesses. For higher number of concurrent reads, files with lower replication factor shows low bandwidth. When remote requests are serviced, the network bandwidth is shared among the requests. With fewer number of replicas and high concurrent access, the contention for network resource of the nodes containing the replica is high, leading to low available bandwidth per access.

5.3.5 Impact of Access Variation

Not all replicas of a file are accessed equally. Whenever a client requests the *NameNode* for accessing a block, the *NameNode* returns the location of all the replicas of the block. The location list is ordered by its proximity from the requesting client. The client checks the availability of a local replica and if it fails to find one, it looks for a rack local, and then a remote rack occurrence of the block. If there are many more concurrent accesses than number of replicas, the scheduler is unable to balance the load to all replicas, because Hadoop performs scheduling on a best effort basis. Figure 5.7 shows the standard deviation in the number of accesses to a block of data with varying number of requests to the data and with varying replication factor. A workload with 3 replicas and 80 concurrent accesses produces very high variance, while increasing the replication produces a more uniform access pattern and thus lower variance. Such uniform access increases the overall throughput because the load in the system is more balanced and the contention for hardware resources is reduced.



(a) 5 GB file with replication factors 1 and 6.



(b) 10 GB file with replication factors 1 and 6.

Figure 5.5 Data distribution in Hadoop with increasing file size and increasing replication factor.

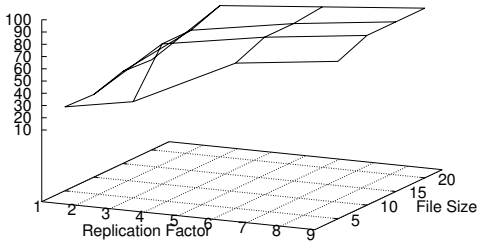


Figure 5.6 Percentage of local tasks scheduled, with increasing file size and increasing replication factor.

5.3.6 Impact of Replication and File Size

Locality increases with the distribution of data in the cluster. Figure 5.5 shows that, as with increasing replication, increasing file size increases the data distribution among the cluster nodes, resulting in reduced contention and increased locality. Figure 5.5(a) shows that data with a replication factor of 1 is available only in one rack, but as the replication increases to 6 we find that data is more evenly distributed among racks. Figure 5.6 shows that with increasing file size and with increasing replication more and more local tasks are scheduled. Previous studies has shown similar behavior with MapReduce benchmarks such as TeraSort [113]. These results are promising as our PPA takes into account such behavior to provide accurate predictions.

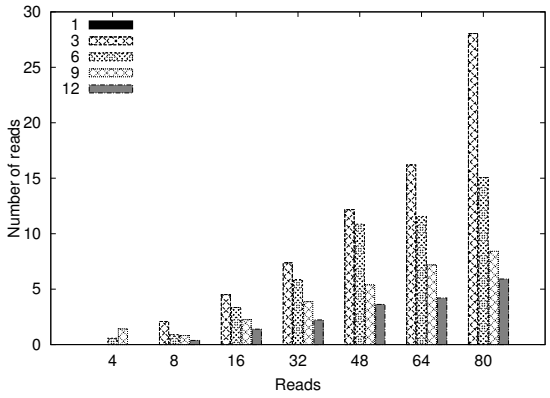


Figure 5.7 Standard deviation in the number of accesses to a block of data while increasing the replication factor and the number of concurrent reads.

Figure 5.8 Comparison of read throughput between Hadoop, Scarlett, AptStore-perf and AptStore.

5.3.7 Fault Tolerance in Hadoop

To study the effect of replication on overall fault tolerance in Hadoop, we first choose an appropriate failure model. We base our probability of data loss model on a previous study [220] and define $P_{data-loss}$ as:

$$P_{data-loss} = 1 - \sum_{f=0}^{n} P_{failure}(n, f) * P_{no-loss}(n, b, r, f), \tag{5.3}$$

$$P_{failure}(n, f) = \binom{n}{f} * p^f * (1-p)^{(n-f)}, \tag{5.4}$$

$$P_{no-loss}(n, b, r, f) = (1 - \binom{f}{r} / \binom{n}{r})^b \tag{5.5}$$

where p is the probability of failure of a single machine, n is the number of machines in the

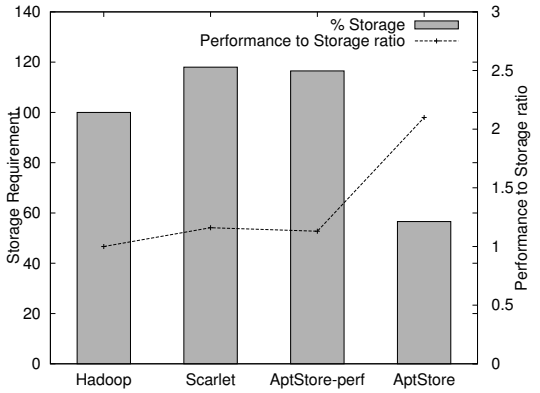


Figure 5.9 Comparison of storage requirement between Hadoop, Scarlett, AptStore-perf and AptStore.

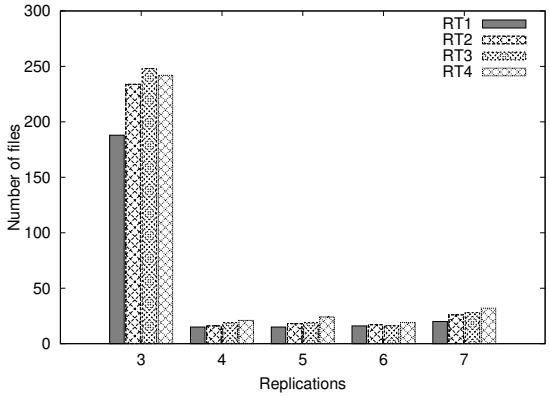


Figure 5.10 Number of files with different number of replicas.

cluster, r is the replication factor of a block, $P_{failure}(n, f)$ is the probability that there are exactly f failures in the cluster and $P_{no-loss}(n, b, r, f)$ is the probability that there is no data loss in the cluster [220]. Moreover, the probability of losing a node in time T, is $1-R$, where R is the reliability of a node or the probability that the node will not fail over the time T. R is defined as:

$$R = e^{-\frac{T}{MTBF}} \tag{5.6}$$

The MTBF of a Hadoop node and an enterprise storage server is three and six years respectively [184, 221]. Based on these values, Table 5.1 compares the probability of data loss per day in HDFS and NAS filers using Equations 5.3. We compare the HDFS cluster with 1000 nodes to a filer with 100 nodes assuming that they can offer the same storage capacity. This is valid assumption given recent trends in storage capacity. Enterprise systems can easily support more than 240 TB of storage [126], while a typical Hadoop node has 12 TB to 24 TB of storage.

Table 5.1 Probability of data loss per day.

File System	Replication Factor	Number of nodes	$P_{data-loss}$ per day
HDFS	3	1000	$6.44 * 10^{-2}$
HDFS	3	100	$1.1 * 10^{-4}$
HDFS	2	1000	0.23
HDFS	2	100	$3.86 * 10^{-3}$
HDFS	1	1000	0.59
HDFS	1	100	$8.7 * 10^{-2}$
Filer	3	1000	$1 * 10^{-2}$
Filer	3	100	$1.42 * 10^{-5}$
Filer	2	1000	$7 * 10^{-2}$
Filer	2	100	$9.7 * 10^{-4}$
Filer	1	1000	0.36
Filer	1	100	$4.4 * 10^{-2}$

Filers can offer probability of data loss with one replica of $4.4 * 10^{-2}$, compared to HDFS with 3 replicas, which has a $6.44 * 10^{-2}$ probability of data loss. Given the high number of disks in a single enterprise storage node, fault tolerance is handled by a RAID controller with a probability of data loss of $4.4 * 10^{-3}$, and hence in this case we decrease the replication factor to 1. Disks are arranged in, for example, a $(10, 2)$ RAID array, which protects from a simultaneous loss of two disks with ten-fold decrease in storage overhead when compared to two replicas. Thus, the use of filers as secondary storage is promising in AptStore and offer a most cost-efficient yet robust solution.

5.3.8 Performance Analysis of AptStore

In the next set of experiments, we evaluate AptStore both in a real system as well as using whole-system simulation.

5.3.8.1 Experiments Using Trace Driven Simulation

For our simulation, we replay Facebook-like traces synthetically generated by sampling historical MapReduce cluster traces. The traces provided by Chen et. al. [34] are one day in duration and contain 24 historical trace samples each 1 hour long.

We use these traces to compare the performance of AptStore with the default Hadoop and

an emulated version of Scarlett [43]. Scarlett is a budget-based Hadoop system that increases throughput by replicating files based on their access patterns. For every file, Scarlett computes the maximum number of concurrent accesses (cf) in a learning window of length TL. Once in every rearrangement period, TR, Scarlett computes desired replication factors for each file as $\max(cf + \delta, 3)$, where δ is the cushion factor against under estimation. In our simulation there were 24 rearrangement periods, i.e., rearrangement happens once every hour and our learning window is the same as that of the rearrangement period. We simulate various factors including the impact of replication, contentions while accessing popular data, and advantages of distribution of data.

The comparisons are made with two variants of our system. The first version, AptStore-perf stores file only in HDFS and the minimum replication factor of both popular and unpopular data is 3, while the maximum factor varies based on the popularity of the data. The second version, AptStore, uses two underlying file systems. The popular files reside in HDFS with varying replication and the unpopular files are pushed to a lower-throughput reliable storage with only one replica. AptStore-perf isolates and measures the performance gains of our techniques, while AptStore provides insight into both the performance and storage efficiency. Similar to Scarlett, our simulation of AptStore rearranges the data once every hour.

Figure 5.8 shows the comparison in the read throughput normalized to default Hadoop. Scarlett produce a 19.57% improvement in performance while AptStore-perf and AptStore produce 23.74% and 18.64% improvement over default Hadoop, respectively. It is important to note that in AptStore, where the popular files are fetched from primary storage and the unpopular files are fetched from secondary storage, only 2.6% of accesses are served by the secondary storage.

In the next experiment, we compare the storage requirement, shown in Figure 5.9, where the required storage is normalized to the case of default Hadoop. Scarlett requires 13% additional storage while AptStore-perf uses only 10% additional storage to achieve the same performance. AptStore achieves the same performance using only 57% of the storage required under standard Hadoop. The replication required for unpopular files in the secondary storage is considerably low when compared to the 200% storage overhead of the primary distributed file system. Figure 5.9 also compares the performance to storage ratio, normalized to default Hadoop and AptStore achieves a significant 2× improvement over default Hadoop. Over the default Hadoop, Scarlett shows a 5% improvement in the ratio and AptStore-perf shows 12.5% improvement.

5.3.8.2 Experiments on a Real Testbed

We implemented AptStore on top of USS [91, 92]. Since all file system requests are handled by USS, the USS audit logs record all the access to the underlying file systems. The Hadoop master node communicates with AptStore, which provides it with a data management and replication strategy. Note that for very large clusters our system can run on a cluster of machines to compute the data placement strategy. AptStore accesses the logs and the PPA assigns a popularity to each file at the end of every reference time. The decision system gives hints to the USS for the appropriate replication policy for the file. Migration is performed through POSIX-like USS file system API, while replication of files in HDFS uses the `setrep` file system API in Hadoop. Our workload is generated from traces mentioned in section 6.3.4.1. We replace the jobs in the trace with `sort`, `grep` and `wordcount` [222]. We believe this approximation is reasonable as the advantage of AptStore is mainly because of read access in the map phase. For our implementation, we adjust the length of the trace and the size of the files to match the size of our test cluster. We did not have access to an enterprise filer, so we used HDFS for all the data, irrespective of their popularity. Our implementation shows that AptStore reduces the execution time of the trace by 21.9% over default Hadoop, with 11.9% increase in storage. Figure 5.9 compares the number of files with different number of replicas. We observe that increasing the replication of 19% of files, results in a performance improvement of 21.9% over default Hadoop. The increase in the replication factor of certain files does not pertain to one single factor, it is based on the combination of factors described in section 5.2.3.

Our evaluations show the analysis of various factors affecting the read throughput and fault tolerance in HDFS and enterprise storage solution. We also show the impact of these factors on the design of AptStore, by comparing the performance of AptStore to Hadoop and Scarlett.

5.3.8.3 Chapter Summary

In this chapter, we presented a dynamic data management scheme for Hadoop for achieving higher throughput and lower storage cost. We observe that managing all Hadoop data in a uniform manner results in increased storage overhead or reduced read throughput. For popular files, default replication is insufficient and leads to decreased throughput. For unpopular files, default replication results in storage inefficiency. We proposed AptStore, a system that

exploits the heterogeneity in access patterns to achieve overall reduction in storage cost and increase in read throughput. We identify various factors that affect the throughput of the system and propose PPA to predict the popularity associated with each file, and use the information to adjust the replication and data placement strategy of the files. Using extensive simulations and a real deployment, we demonstrated that AptStore data management scheme increases the read throughput by 23.7%, reduces overall storage utilization by 43.4%, and results in speeding up the studied jobs by as much as 21.3%.

Chapter 6

Maximizing the Benefits of Hierarchical Storage in Hadoop

One of the major challenges in Hadoop is to design and adapt the storage and I/O infrastructure to incorporate the exponentially growing data volumes in an economical fashion. This is non-trivial, especially as the bandwidth provided by the cluster networking infrastructure is growing an order of magnitude faster than the I/O bandwidths of hard disk drives (HDDs) [31]. In a typical large-scale Hadoop deployment, the intra-rack and inter-rack network has a bandwidth 200× and 400× that of the disk bandwidth [31], respectively, underscoring the bottleneck due to the bandwidth of the storage devices. Solid-state drives (SSDs) can help mitigate this performance gap but at a higher cost for the storage capacity.

A Hadoop job involves two types of I/O: (i) the data that resides in HDFS and serves as the input and the output of a Hadoop program; and (ii) the intermediate data that is generated during application execution and resides in the local file system of the node on which it is created. Placing the HDFS data and/or the intermediate data in SSDs can improve Hadoop jobs performance [40, 41]. Recent efforts [38, 39] have shown that SSDs are a viable alternative to HDDs for Hadoop I/O. However, replacing all HDDs in a Hadoop deployment with SSDs is not economically viable.

A promising trend observed in recent analysis [42,43] is the significant heterogeneity in HDFS I/O access patterns, which enables classification of data as 'hot' and 'cold' [223] based on its usage characteristics. Recent research has shown that adding an SSD tier to store the hot HDFS data can improve I/O performance [43, 224, 225]. While our analysis of synthetic Facebook logs [211] reveal that 30% of the total I/O is that of intermediate data, not much research is done into studying the impact of improving the intermediate I/O performance of an application. We also observe that improving the latency of HDFS or intermediate I/O may not result in an increase in the execution time for every application. Storing selected

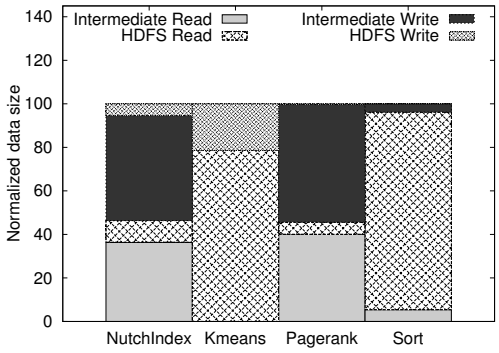


Figure 6.1 Observed variation in data access pattern across applications. All access values are normalized to the total I/O accesses of an application.

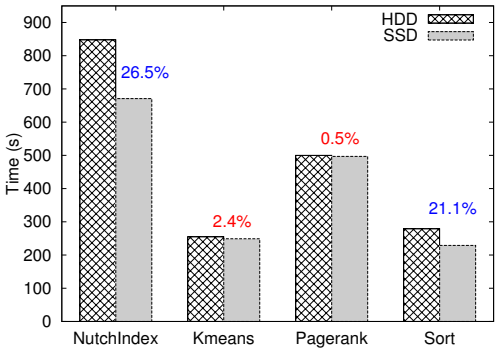


Figure 6.2 Observed variation in applications execution time due to the use of SSDs and HDDs to store and retrieve both HDFS and intermediate data.

data that impacts execution time in the SSD tier can improve the performance of the overall system at an affordable cost. This approach is promising, but introduces the challenge of identifying the appropriate data to be placed in the SSD tier, effectively managing the distribution of data among different tiers and selecting a tier for servicing I/O requests with the goal of improving application's execution time. In this paper, we address this challenge and explore the design space of incorporating a flash tier that will benefit both the intermediate I/O and the HDFS I/O in Hadoop.

To study the impact of storage technology on overall application execution time, we ran four Hadoop/MapReduce applications[1], namely *NuthchIndex*, *Kmeans*, *PageRank*, and *Sort*, on a eight-node test cluster[2]. Figure 6.1 shows the observed variation in disk access patterns for the studied applications. For *NuthchIndex* and *PageRank*, more than 85% of the data accesses are for intermediate data. Conversely, for *Kmeans* and *Sort*, more than 90% of the data accesses are for HDFS. Similarly, the impact of the underlying storage technology on the total job execution time varies from application to application. Figure 6.2 shows the observed variation in application execution time due to the use of SSDs and HDDs to store and retrieve both HDFS and intermediate data. We observe that for *NutchIndex* and *Sort*, SSDs can contribute a performance gain in the execution time of up to 26.5% and 21.1%, respectively, while for *Kmeans* and *PageRank*, the performance gain is less than 3%. Both *NutchIndex* and *PageRank* access approximately 14 GB of intermediate data and total I/O accesses of 15 GB and 17 GB, respectively, but the benefits of using SSDs are different; the former shows 26.5% performance gain, while the later shows only 0.5%. This experi-

[1]We selected these applications to highlight the variation in access patterns.

[2]Each node consists of two 2.8 GHz quad-core Intel Xeon processors, 8 GB of RAM, one PCIe SSD and one SATA HDD.

ment shows that I/O access patterns can vary widely across applications and the impact of storage technology on application execution time is not constant. This reinforces the need for an application-attuned storage management system for Hadoop. Identifying applications that will benefit from accessing HDFS and/or intermediate data from the SSD tier and efficiently managing the storage tiers can improve the performance while potentially reducing the number of required SSDs and therefore minimize the cost of using SSDs in Hadoop.

We design DUX, a system that reduces the overall I/O latency and execution time of applications by introducing *application-attuned storage management* in Hadoop. DUX employs a tiered storage system with two tiers designed to better match the heterogeneous Hadoop I/O access patterns. The first tier is a *fast SSD tier*[3] that aggregates the SSDs provisioned in each node, and serves as a cache for a *secondary HDD tier* comprising of HDDs. The observation driving DUX design is that performance gains from using SSDs for either HDFS data or intermediate data are dependent on an application's I/O access patterns. Hence, in a heterogeneous environment where each node has both SSDs and HDDs, the choice between serving HDFS data, intermediate data or both from the SSD should be based on the application's I/O profile. To this end, the key contribution of DUX is that it profiles application I/O behavior on different storage configurations and proposes an appropriate storage configuration for future application execution. DUX provides a holistic solution for effectively orchestrating the SSD tier by performing three major functions: (i) it provides a platform that enables the SSD tier to be shared by the popular HDFS and intermediate files; (ii) it predicts the impact of I/O accesses on execution time and chooses an appropriate tier for storing the intermediate data; and (iii) it prefetches the input data into the SSD tier for jobs waiting in the job queue, if the data has not been selected for prefetching by the popularity predictor.

A concern in employing SSDs as a cache is that they have limited erase cycles, and may affect the MTTF. We stress that incorporating SSDs to form a caching tier is not unique to our approach, and state-of-the-art works [228, 229] have also purported the same. Moreover, numerous SSD optimization approaches are available [208, 209] to remedy this, which can be leveraged in DUX. SSD endurance is orthogonal to our design and DUX is useful even when no SSDs are used but different kinds of storage technologies, such as, RAM-based file system (ramfs) [230] and SCM [231], are employed as cache.

Specifically, DUX makes the following contributions:

[3]This tier can be replaced by any fast storage technology, such as, RAMDisk [226, 227]

- We present a detailed quantitative study of factors that affect the I/O of Hadoop applications, the impact of improving the access latency of HDFS and intermediate data on execution time, the resulting I/O access patterns, and the rate of intermediate data generation with respect to the application.
- We realize enhancements for HDFS to support the proposed SSD caching tier and data prefetching between the proposed tiers.
- We design and implement DUX to track, record and analyze application behavior on different storage configurations, and use the knowledge to dynamically propose appropriate storage tiers for HDFS and intermediate data for the submitted jobs.
- We validate DUX design and techniques therein using in-depth simulations and a real Hadoop deployment.

6.1 Design of DUX

In this section, we present the design of DUX and how we address the challenges faced therein.

6.1.1 DUX Overview

The design of DUX is inspired by three factors. First, the access rate and the number of accesses vary for each file in HDFS [43], leading to popularity skewness over a smaller subset of files. Second, the I/O characteristics of applications vary, i.e., the size of HDFS data accesses, and the intermediate data generated, will vary across applications. Third, application execution time and I/O latency of HDFS and intermediate data is correlated in a similar fashion for all applications. Considering these factors, the goal of DUX is to improve the overall read throughput and storage efficiency of Hadoop clusters. This is achieved by exploiting the heterogeneity in access patterns and application behavior for effectively using the SSD tier. To this end, we foresee Hadoop clusters comprising nodes that have attached SSDs. DUX divides different storage types into tiers, i.e., HDD tier and SSD tier, and enables effective utilization of the SSD tier by using it as a cache. We propose three key optimizations in DUX:

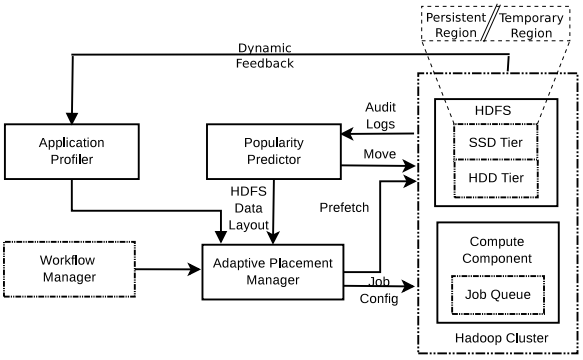


Figure 6.3 DUX architecture overview.

- Access pattern based input data placement;
- Application characteristics based intermediate data placement; and
- Prefetching (of input data) from the HDD tier to the fast SSD tier as needed.

Figure 6.3 illustrates the main components of DUX and their interactions. The first component is the *Application profiler*, which profiles applications that are deployed on the target Hadoop clusters. It test runs the representative applications on all possible storage configurations of a cluster. A *storage configuration* is a per-job property that specifies which part of application data is placed on a particular tier, e.g., input data on HDD and intermediate data on SSD, or both types of data on SSD, etc. For each application, the profiler records essential information, such as, the volume of data read/written in different phases and the completion time of these phases under each configuration. This profiling information is then used to guide the various components of DUX.

The *Popularity Predictor* component dynamically keeps track of the access counts of input data files in HDFS. Based on this information, the component periodically makes a prediction for the popularity of each data file in the upcoming interval and instructs HDFS to move the popular files into the SSD tier if needed. The *Adaptive Placement Manager* component analyzes the jobs that are submitted to the workflow manager, and based on the input from the application profiler, decides appropriate storage configurations for each job. If a job performs better with low latency storage devices, the Adaptive Placement Manager changes the job configuration of the data file and prefetches the required HDFS data into the SSD tier.

6.1.2 Enabling Technology: hat*S*

hat*S* [232] is an enhancement to HDFS that provides heterogeneity-aware tiered storage in Hadoop. hat*S* logically groups all storage devices of the same type across the nodes into an associated 'tier'. A deployment has as many tiers as the types of storage devices used. A node with multiple types of devices is a part of multiple tiers. For instance, if a deployment consists of nodes with an SSD and an HDD, all SSDs across the deployment will become part of a SSD tier, and similarly all HDDs will form a HDD tier. By managing tiers individually, hat*S* is able to capture the heterogeneity in hardware and exploit it to achieve high I/O performance. While HDFS considers only network-aware data placement and retrieval policies, hat*S* proposes additional policies to replicate data across tiers in a heterogeneity-aware fashion. This enhances the utilization of the high-performance storage devices by efficiently forwarding a greater number of I/O requests to the faster tier, thus improving overall I/O performance.

We leverage hat*S* to provide an enhanced HDFS component for the system. We employ the default Hadoop data placement policy but ensure that the data is placed only in the HDD tier by default. Also, in order to avoid network contention during accesses and to provide tolerance against node failure, we ensure that whenever a block is moved to the SSD tier, it is not replicated at the node that already stores it. The data retrieval policy that we employ always accesses the data from the fastest available tier. One concern is that the tier-aware placement may be violated in the case of node failures. To overcome this, we utilize hat*S* monitoring daemon at the NameNode to ensure that any re-replication is done on the same tier from which the data is lost. Moreover, we also provide APIs that are used by the Adaptive Placement Manager to enable data movement between tiers.

6.1.3 Enabling Technology: Popularity Predictor

The Popularity Predictor uses HDFS audit log information to determine the popularity of a file in HDFS in order to proactively fetch the popular data from the HDD tier into the persistent region of the SSD tier. This is an effective approach for identifying hot data as also shown in the previous work [224, 225]. The popularity predictor periodically analyzes the access patterns of each file and predict the file's expected popularity value for the next interval. The length of each interval is called Reference Time (RT). The choice of RT is critical; a very large RT can result in a stale SSD cache tier, whereas a small RT can increase network

traffic. Previous work [43, 224] suggests that an RT between 12 and 24 hours is sufficient. When a file is created, its popularity is initialized to average file popularity observed in the system. For each access to a file, its popularity is increased by one. Similarly, whenever a file is deleted, the popularity of other files is updated based on the popularity of the deleted file. When a popular file is deleted, the popularity of other files in the system increases. Conversely, when an unpopular file is deleted, the popularity of other files decreases. After the accesses to all files in the system are processed, the popularity value of a file for the most recent access is decreased by the mean increase in the popularity of the file during that RT. This is done to makes sure that the popularity of the file does not grow arbitrarily. The mean increase in popularity is a fraction of increase in popularity during RT over the set of all files in the system. Finally, we compute the predicted popularity of the file, for the next RT by linear extrapolation. We assume that the rate of change of popularity for the next RT will be same as the rate of change of the current RT. Such linear extrapolation based prediction ensures that cold data that becomes hot after a while will also be stored in the SSD tier. The files in HDFS are sorted in the order of their predicted popularity, and popular files are loaded into the SSD tier, until the persistent region is full.

6.1.4 SSD Capacity Management

In DUX, the SSD tier is used for three types of cached data: the intermediate data, the popular input data, and the unpopular prefetched input data. To enable configurable sharing of the SSD tier between different data types, the cumulative storage capacity of SSDs aggregated across all nodes is virtually split into a *persistent region* and a *temporary region*. This is inspired by pattern-based OS caching approaches [233, 234]. Files that are popular are placed in the persistent region, while the temporary region is utilized for the intermediate data and for prefetched HDFS data that is not yet popular but will be used by jobs that are next in the job queue. Input files for jobs that are scheduled for later execution are also prefetched into the temporary region. The size of the persistent region is configurable, and should be selected based on the utilization of the cluster. For heavily-loaded clusters, a large volume of intermediate data is generated, therefore, a large capacity should be allotted for the temporary region. A smaller persistent region means a smaller number of popular files will be moved into the SSD tier by the Popularity Predictor. Consequently, the amount of data prefetched by the Adaptive Placement Manager will increase, causing an increase in the network traffic that must be sustained by the network infrastructure. In this way, splitting

the SSD capacity allows the users to tune the storage performance based on the provisioned network capacity.

6.1.5 Application Profiler

DUX employs an Application Profiler that performs statistic profiling of applications to better guide its data placement and resource allocation. Applications are executed in a test environment on all possible storage configurations and metrics are recorded related to the I/O profile of applications, such as, amount of input data read from HDFS, amount of intermediate data generated, job completion time on each configuration, time spent in the map, sort/shuffle and the reduce phases on each configuration etc. Each configuration is tested multiple times to record average metric values. Throughout this profiling, we treat an application as a black box, i.e., we do not analyze the application code and the profiled information are based only on the execution history.

We argue that statistic profiling of workloads prior to production execution is ideal for this scenario as the collected statistics do not have interference from other jobs. However, a dynamic feedback loop is also needed to approximate the characteristics of new jobs that are not profiled but submitted to the cluster after the statistic profiling has been done.

The Application Profiler maintains a database of the results from statistic profiling and the execution history of applications. This database includes information, such as, a list of recently completed applications, associated execution times, input data size, and the size of intermediate data generated during prior runs of the same application (both from statistic profiler and dynamic feedback). The database is queried by the Adaptive Placement Manager when it needs to choose a storage configuration for the submitted job.

To derive our profiling techniques, we studied 10 representative MapReduce applications from HiBench [235], which cover a wide range of workload behavior, such as, batch processing, iterative jobs and interactive querying. This is motivated by the previous research [52, 93], which has shown that MapReduce workloads are predictable for their behavior and that the number of different job types is small. Studying a range of test applications on target clusters enables us to determine the initial job schedule in a multi-cluster deployment.

6.1.6 Popularity Predictor

The Popularity Predictor uses the HDFS audit log information to determine the popularity of a file in HDFS in order to proactively fetch the popular data from the HDD tier into the persistent region of the SSD tier. This is an effective approach for identifying hot data as also shown in previous work [224,225]. The popularity predictor uses Algorithm 2 to analyze the access patterns of each file after periodic time intervals and predict the file's expected popularity value for the next interval. The length of each interval is called Reference Time (RT). The choice of RT is critical; a very large RT can result in a stale SSD cache tier, whereas a small RT can increase network traffic. Previous works [43, 224] suggests that an RT between 12 and 24 hours is sufficient.

When a file is created, its popularity $P_1(f)$ is initialized to average file popularity observed in the system, $AVG(P)$. For each access to the file, the popularity of the file is increased by one. Similarly, whenever a file is deleted, the popularity of other files is modified based on the popularity of the deleted file. When a popular file is deleted, the popularity of other files in the system increases. Conversely, when an unpopular file is deleted, the popularity of other files decreases. After the accesses of all files in the RT are processed, the popularity value $P_i(f)$ of a file f for the most recent access i is decreased by the mean increase in the popularity of the file f during that RT. This is done to makes sure that the popularity of the file $P(f)$ does not grow arbitrarily. The mean increase in popularity is a fraction of increase in popularity, IP, during RT over F, the set of all files in the system. Finally, we compute the predicted popularity of the file, $P_{predicted}(f)$, for the next RT by linear extrapolation. We assume that the rate of change of popularity for the next reference time will be same as the rate of change of the current reference time. Such linear extrapolation based prediction ensures that cold data that becomes hot after a while will also be stored in the SSD tier. The files in HDFS are sorted in the order of their predicted popularity, and popular files are loaded into the SSD tier, until the persistent region is full.

When a file is created, its popularity $P_1(f)$ is initialized to average file popularity observed in the system, $AVG(P)$. For each access to the file, the popularity of the file is increased by one. Similarly, whenever a file is deleted, the popularity of other files is modified based on the popularity of the deleted file. When a popular file is deleted, the popularity of other files in the system increases. Conversely, when an unpopular file is deleted, the popularity of other files decreases. After the accesses of all files in the RT are processed, the popularity value $P_i(f)$ of a file f for the most recent access i is decreased by the mean increase in the popularity of the file f during that RT. This is done to makes sure that the popularity of the

Input : HDFS audit logs
Output: HDFS files with their new popularity based on the access count.

F is the set of files in the file system;
foreach *access $i+1$ to the file $f \in F$ in RT* **do**
    if *i==0* **then**
        $P_{i+1}(f) \leftarrow AVG(P)$;
    end
    else
        $P_{i+1}(f) \leftarrow P_i(f) + 1$;
    end
    $IP = IP + P_{i+1}(f) - P_i(f)$;
end
foreach *deletion of the file f in F* **do**
    $IP = IP + AVG(P) - P(f)$;
end
$MIP \leftarrow \frac{IP}{size(F)}$;
foreach *file f in F* **do**
    $P_i(f) \leftarrow P_i(f) - MIP$;
    where i is the most recent access to the file f. $P_{predicted}(f) \leftarrow P_i(f) + (P(f) - P_i(f))$;
    $P(f) \leftarrow P_i(f)$;
end

Algorithm 2: Algorithm used by the Popularity Predictor.

file $P(f)$ does not grow arbitrarily. The mean increase in popularity is a fraction of increase in popularity, IP, during RT over F, the set of all files in the system. Finally, we compute the predicted popularity of the file, $P_{predicted}(f)$, for the next RT by linear extrapolation. We assume that the rate of change of popularity for the next reference time will be same as the rate of change of the current reference time. Such linear extrapolation based prediction ensures that cold data that becomes hot after a while will also be stored in the SSD tier. The files in HDFS are sorted in the order of their predicted popularity, and popular files are loaded into the SSD tier, until the persistent region is full.

6.1.7 Adaptive Placement Manager

The task of the Adaptive Placement Manager is to: (i) decide the appropriate storage for each submitted job by querying the Application Profiler, (ii) update the job-associated storage configuration as needed, and (iii) (if required) prefetch unpopular input data into the SSD tier while the job is waiting in the job queue. Whenever a job is submitted, the Adaptive Placement Manager queries the Application Profiler to access critical information about the job, i.e., the job completion time, the time spent in map and shuffle phases on different

storage configurations, and the predicted space requirements for the input and intermediate data. The Adaptive Placement Manager then uses the information to decide the appropriate storage configuration for the job. If the choice of input storage is SSD tier, the Manager accesses HDFS data layout from performance predictor to check if the input data is already available in the SSD tier (i.e., it is popular data). If not, it prefetches the data into the SSD tier. If the SSD tier is the choice of storage for intermediate data, we make sure that the tier has the requisite space. For this purpose, we linearly extrapolate the input data size to predict the available space. Considering potential prediction errors, there is a buffer space in the temporary region to ensure that the jobs do not fail due to missing the required data. The Adaptive Placement Manager also has a *StorageCounter* that keeps track of the storage utilized in the SSD tier. Whenever some HDFS data is prefetched or intermediate data is configured to be stored in the SSD tier, the counter is incremented and then decremented after the job associated with the data completes. *StorageCounter* enables the Adaptive Placement Manager to evict unnecessary data from the SSD tier and to plan the storage configuration for jobs waiting in the job queue.

6.1.8 Discussion

DUX can co-exist with the master component of each cluster or in a separate node. The computational overhead of DUX is small as after initial processing, the requisite algorithm is run once for every new submitted job, and entails a database lookup and execution of the update steps. The Adaptive Placement Manager predicts the configuration of the job by accessing the application profiler, which is based on the job execution traces that are collected as a background process in each node. The performance predictor predicts the popularity of data files by linearly processing ($O(n)$, where n is the number of files) the Hadoop file system audit logs. Moreover the prediction is done every RT, i.e., 12 to 24 hours, further amortizing the associated overhead.

Hadoop performance is sensitive to network bandwidth, particularly during the shuffle phase that involves moving large amounts of data across the network. The Performance Predictor rearranges the data after every RT and the Adaptive Placement Manager prefetches input data into the SSD tier as needed. This entails additional network overhead. Data movement across tiers during the network-intensive shuffle phase may adversely affect the performance of executing jobs. To balance the bandwidth consumption, HDFS employs multi-location replication [43], where the data to be moved across tiers is read from multiple sources thereby

spreading the traffic across nodes. To remedy this, the Performance Predictor's data rearrangement is made a low priority background process that yields to the shuffle phase to avoid negative performance impact. The approach, however, may not be applicable when prefetching, as that handles data that is needed by the jobs in the queue. An approach that can mitigate the problem is to prefetch for jobs further down the queue in advance, provided enough capacity is available in the SSD tier. Finally, placement of intermediate data does not add to any additional network overhead, as the network contention will be the same, irrespective of the storage configuration.

In summary, profiling the characteristics of applications help DUX better understand the behavior and I/O characteristics, as well as the dependence of the I/O on the execution time of the studied applications. Using this information, the Adaptive Placement Manager is able to estimate the required storage space for the applications to run, check for the availability of resources, and modify associated job configuration files accordingly, before submitting the jobs. By better matching jobs with appropriate tiered storage, DUX is expected to yield higher overall performance and achieve better system efficiency, as observed in our evaluation (Section 8.2).

6.2 Implementation of DUX

In this section, we describe our implementation of DUX and related HDFS enhancements. We have implemented a proof-of-concept Popularity Predictor and an Adaptive Placement Manager in Python. We used the *SAR* tool [210] to collect job execution traces containing information about storage configuration of applications. For statistics profiling, we configure HDFS and the intermediate storage to different storage devices to collect separate statistics for input and intermediate data. We also parse Hadoop logs to determine timestamps associated with the start and finish time for the given applications, which are then used to separate execution information for each application. The Application Profiler uses a MySQL database instance to store the collected application information as well as the associated resource utilization.

To enable HDFS to support the caching tier, hat*S*—that serves as our enabling component—configuration files are extended and modified to enable SSD and HDD tiers. To support data placement that will enable prefetching, we modify the NameNode's *ReplicationTargetChooser*. A list of nodes is chosen from the *NetworkTopology* structure that provides

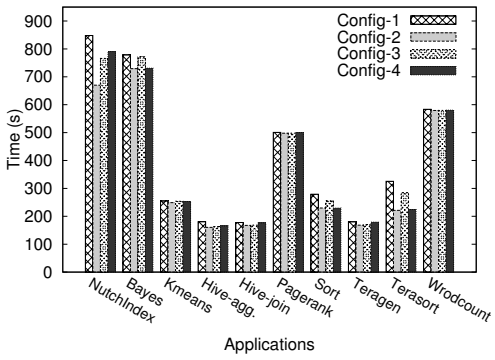


Figure 6.4 Application completion times under *Config-1*, *Config-2*, *Config-3* and *Config-4*.

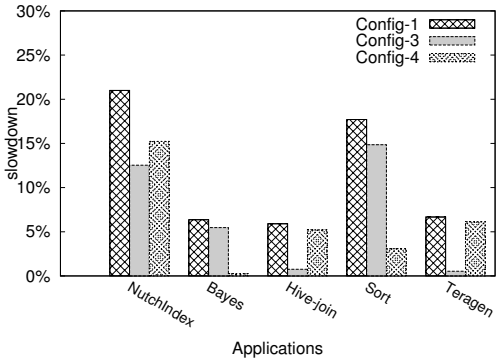


Figure 6.5 Increase in completion times under *Config-1*, *Config-3* and *Config-4* as compared to *Config-2*.

information about various racks and tiers in the cluster (*clusterMap*). Similarly, we modify *NetworkTopology* and *DataNodeDescriptor* in hat*S* to realize appropriate data retrieval policy as described in Section 6.1.

For the Popularity Predictor to load the popular files into the SSD tier and Adapter Placement Manager to prefetch the data from HDD into the SSD tier, we use the *boolean moveToSSDTier (String filename, number_of_replica)* and *boolean moveToHDDTier (String filename, number_of_replica)* APIs provided by hat*S* [232]. For the Adaptive Placement Manager to modify the location of the intermediate data, we set the location of the *mapred.local.dir* to the appropriate storage device. This property is set in command line using the `-D` switch. We also implement a plugin that can modify the configuration for Apache Oozie [85] workflow manager. The components interact as described earlier to realize DUX.

6.3 Evaluation of DUX

In this section, we present the evaluation of DUX using a real deployment on a 9-node Hadoop cluster, as well as using trace-driven simulation that executes synthetic Facebook workloads [34]. We first study the characteristics of 10 representative Hadoop applications on SSD and HDD storage configurations. Next, we evaluate the impact of our HDFS enhancements, performance prediction and adaptive placement. Finally, we compare the overall performance of DUX against the existing application-oblivious storage placement strategy.

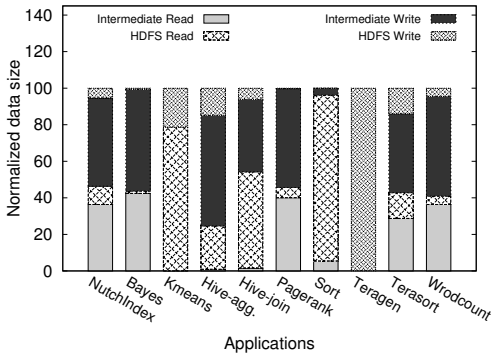


Figure 6.6 Variation in data access pattern across applications. All access values are normalized to the total I/O access in the application.

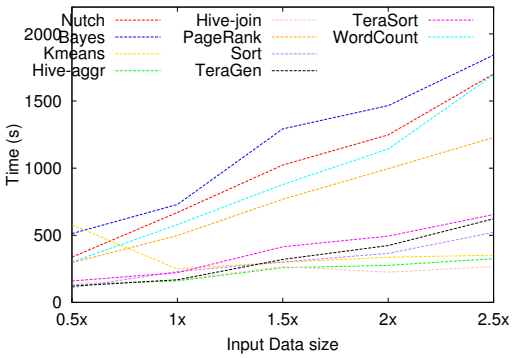


Figure 6.7 Effect of increasing the input data size on application completion time.

Table 6.1 Specifications of storage devices used in our tests.

Device Type	Write BW	Read BW	IOPS	# of Devices
PCIe SSD	245 MB/s	533 MB/s	70k	3
HDD	46 MB/s	61 MB/s	3.5k	27

6.3.1 Experimental Setup

Our testbed consists of a master node and eight worker nodes. Each node has two 2.8 GHz quad-core Intel Xeon processors, 8 GB of RAM, and one SATA HDD. The HDDs are 500 GB 7200 RPM Seagate Barracuda ES.2 drives. In addition to HDDs, each worker node is provisioned with an OCZ RevoDrive series PCIe 128 GB SSD. Table 6.1 shows the specifications of the storage devices. In our setup, all DataNodes contribute to both the SSD tier and the HDD tier. The nodes are connected using a dedicated 10 Gbps InfiniBand switch. Each worker node is configured with six map slots and two reduce slots so that all available cores on a node are used. The benchmark applications are mostly map intensive, so there are more map slots than reduce slots. The master node runs both the JobTracker and NameNode for all experiments, and all the worker nodes contribute to both TaskTracker and DataNode. The replication factor is fixed at the default value (i.e., three), and the block size used in our evaluation is 64 MB.

6.3.2 Benchmark Applications

We have used 10 applications from well-known Hadoop HiBench Benchmark Suite [235] for evaluating DUX. These applications represent batch processing jobs, iterative jobs and inter-

active querying jobs. Table 8.2 lists these applications, and for each summarizes parameters such as the input and output data size, and the number of mappers and reducers.

6.3.3 Application Analysis

In our first set of experiments, we analyze the performance of our benchmark applications under four different storage configurations shown in Table 8.1. The results presented below are average of five executions; the standard deviation across the executions was observed to be negligible.

6.3.3.1 Observing Non-Uniform Effect of Storage on Performance

Figure 6.4 shows the performance of our test applications under storage configurations of Table 8.1. We observe that under *Config-2*, jobs perform 12% faster on average over *Config-1*. Here, *NutchIndex*, *Bayes*, *Sort* and *TeraSort* show an average performance improvement of 19.7%, while the other applications show an average performance improvement of only 2.9%. This observation highlights the differences in benefits that different applications can achieve from the exclusive use of the SSD tier. Moreover, by monitoring the disk accesses we find that *NutchIndex*, *Bayes*, *Sort* and *TeraSort* contribute 60% of the total disk usage under *Config-2*, while all the other applications only contribute 40% aggregately. Therefore, if we were to place the input and intermediate data for only these four applications in the SSD tier and use the HDD tier for all other applications, we will achieve an average performance improvement of 11% under *Config-1*.

Table 6.2 Representative 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
Bayes	128 MB	256 KB	4.5 GB	16	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
Sort	3 GB	11.5 GB	3 GB	64	8
TeraGen	–	–	15 GB	16	0
TeraSort	15 GB	15 GB	15 GB	249	8
WordCount	12 GB	30 GB	12 KB	102	8

Table 6.3 Different storage configurations used in statistical profiling.

Configuration	HDFS Data	Intermediate Data
Config-1	HDD	HDD
Config-2	SSD	SSD
Config-3	SSD	HDD
Config-4	HDD	SSD

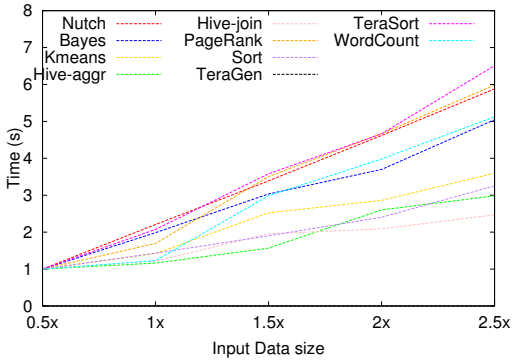


Figure 6.8 Effect of increasing the input data size on intermediate data generation.

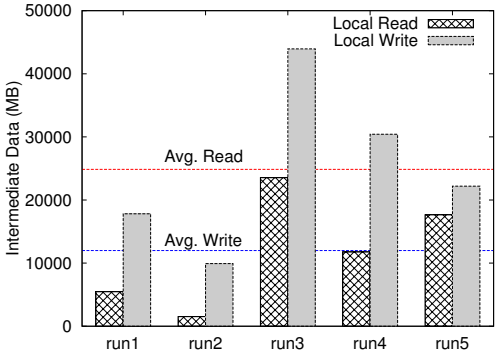


Figure 6.9 Intermediate data generated by Grep for different queries on same input data set.

Comparing the performance of our benchmark applications under *Config-3* and *Config-4* helps us determine the impact of the performance of input or intermediate storage on the overall performance of an application. We highlight this in Figure 6.5 by showing the *percentage increase* in the completion times (i.e., slowdown) of *NutchIndex*, *Bayes*, *Sort*, *TeraGen* and *TeraSort* under *Config-3* and *Config-4* against the respective completion times under *Config-2*. *NutchIndex* experiences an identical slowdown in both *Config-3* and *Config-4*, which shows that it needs both its input and intermediate data in the SSD tier for optimal performance. *Bayes*, *Sort* and *TeraSort* only experience a slowdown under *Config-3*, indicating that their performance depends entirely on intermediate data storage. Finally, *TeraGen* depends entirely on the performance of its output storage. This analysis validates the need for the proposed Adaptive Placement Manager.

6.3.3.2 Variation in Data Access Pattern

In the next set of experiments, we study the variation in access pattern across different applications. Figure 6.6 shows the results. For applications such as *NuthchIndex*, *Bayes*, *Pagerank*, *TeraSort* and *Wordcount*, more than 50% of the data accesses are for intermediate data. Conversely, for applications such as *Kmeans*, *Hive*, *Sort* and *TeraGen*, more than 50%

of the data accesses are for HDFS. Both *NutchIndex* and *PageRank* access approximately 14 GB of intermediate data and total I/O accesses of 15 GB and 17 GB, respectively, but the benefits of using SSDs are different; the former shows 26.5% performance gain, while the later shows only 0.5%. This analysis validates the claim that applications with similar access patterns show variation in execution time, thus forcing the need for an application-attuned storage configuration. Our approach will ensure that the SSD storage tier services only those applications that will benefit from the low-latency storage device, thus effectively maximizing the use of the available SSD space.

6.3.3.3 Impact of Data Size

In the next set of experiments, we study the impact of increasing input data size on the performance of the studied applications. Figure 6.7 shows the completion time of applications under varying input data sizes for *Config-4*. We increase the respective input sizes shown in Table 8.2 from 0.5× to 2.5×. We find that although the increase in the completion time is linear, the rate of increase is not the same across all applications, e.g., *Hive* takes 2× time to process 5× more data, whereas, *WordCount* takes 6× time. Figure 6.8 shows the effect of increasing the input data size on intermediate data generation. Similar to completion times, we observe a linear increase in the input data generation. The scaling factor for the intermediate data generation and the completion time is unique per application. Understanding this enables us to better estimate the time and resources required by the application to execute on a particular storage configuration with a given data set size. We note that a similar performance-data size pattern is also observed under other hardware configurations, though the rate of increase in application completion time and the input data generation varied across the storage configurations by a linear factor.

6.3.3.4 Limitation of DUX

Throughout the profiling process, we treat the application as a black box. Our prediction on intermediate data generation and execution time is based on multiple executions of the application for varying data sets. Although our approach is effective for some applications such as *Sort*, *NutchIndex*, *WordCount*, *TeraSort* and *KMeans*, it can not be generalized for all possible applications.

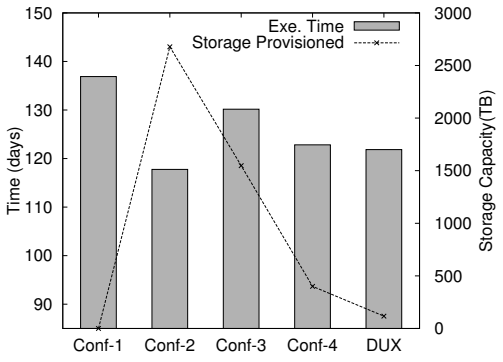


Figure 6.10 Performance variation observed under the studied configurations and associated storage capacity.

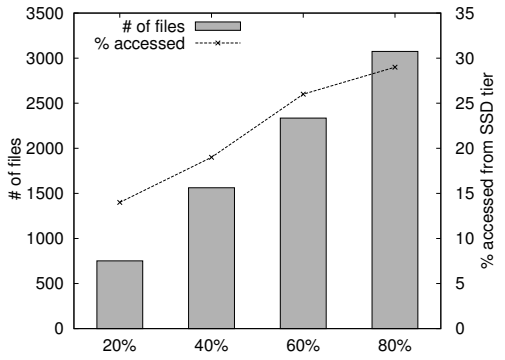


Figure 6.11 Effectiveness of the Performance Predictor with increasing persistent region.

Figure 6.9 shows the intermediate data generated across different executions of Grep on the same data set with different search queries. We observe that the intermediate data generated is not constant across all runs and the standard deviation is 9,000 Bytes and 17,000 Bytes for read and write, respectively. DUX will benefit from static program analysis techniques [236] designed for Hadoop applications that can provide hints about the behavior and the characteristics of the data. While there is an emerging body of work in the area of static analysis of Hadoop applications [237, 238], there exists no work that is directly applicable in the targeted context. Moreover, while complementary, such work is beyond the scope of DUX.

6.3.4 Performance Analysis of DUX

In the next set of experiments, we evaluate the overall performance of DUX using a real system as well as through whole-system simulation.

6.3.4.1 Trace Driven Simulation

For our simulation, we replayed Facebook-like traces that are generated synthetically by sampling historical MapReduce cluster traces. These are representative of traces generated from a Facebook cluster from October 2010 for a period of one and half months and are provided by Chen et. al. [34]. The available workload for a one day duration and contains 24 historical trace samples, each of 1 hour long. The traces contain the record of accesses to approximately 18,000 files (along with filenames) constituting a total input size of 1548 TB.

The size of the shuffle data generated is 375 TB and the output data generated is 755 TB. 55% of files are accessed more than once while the other 45% are accessed only once. The highest number of accesses to any file is 721. These files are accessed by 25,500 jobs, with 1075 jobs accessing the files every hour.

In our simulation we set $RT = 1$ hour, i.e., the performance predictor ran once every hour, generating a new set of popularity predictions and moving the most popular files to the SSD tier. This creates a new HDFS layout on every interval. Our simulation incorporates various factors including the impact of replication, contentions while accessing a data file, and advantages of data distribution along with those observed in Section 6.3.3.

Figure 6.10 shows the completion time across the four fixed storage configurations from Table 8.1 using DUX. We observe that *Config-2* is 14% and 3% faster than *Config-4* and *Config-3*, respectively. Whereas, DUX is 10% and 6% faster than *Config-1* and *Config-3*, respectively. We note that although DUX is 5% slower than *Config-2*, DUX uses 5.5× less storage than that under *Config-2*.

Next, we simulate the effect of increasing the size of persistent region in the SSD tier on the performance predictor. Figure 6.11 shows the variation in the number of cached files and the percentage of accesses for the SSD tier as the size of persistent region is increased. We observe that with increasing persistent region size, a larger number of input files are stored and accessed from the SSD tier. Figure 6.12 shows the effect of increasing the size of temporary region and its impact on application completion time and network traffic. We observe that increasing the temporary region size increases the performance of the applications but at an increased network overhead. This is because the performance predictor is only able to load an increasingly smaller subset of files into the persistent region, which results in more data being prefetched, consequently increasing the network traffic. In clusters where network capacity can potentially become a bottleneck, using large temporary region sizes may not be a viable option.

6.3.4.2 Implementation Test

In order to validate the results from our simulation, we perform tests on a real cluster described in Section 6.3.1. Our workload is generated from traces described in Section 6.3.4.1 and a slice of this workload (lasting for approximately 3 hours) is executed. We replace jobs in the trace with the benchmark applications (Table 8.2). We note that our implementation

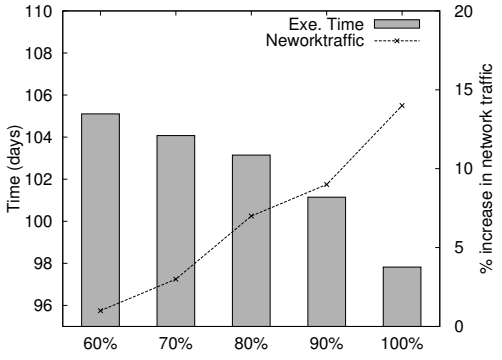


Figure 6.12 Observed performance and network traffic under varying persistent region.

Figure 6.13 Observed performance benefit under DUX on a real testbed.

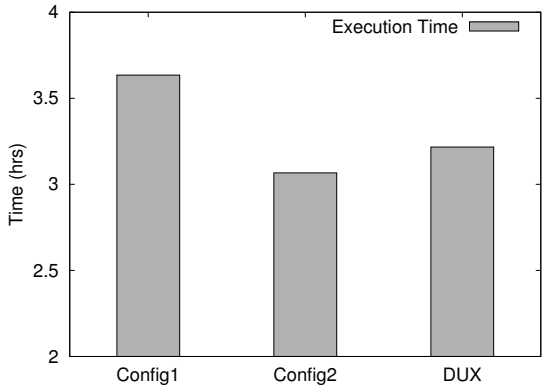


Figure 6.14 Observed performance benefit under DUX on a real testbed.

results do not include the Performance Predictor component, as the workload length is too small to observe any significant advantages. For the same reason, we also limit our SSD capacity to be 10% of the available storage. DUX was run on an independent node (so as to avoid influence of Predictor's computations and Profiler's database on the running job) and communicated with the master node to modify the storage configurations. Figure 6.14 shows the results. We observe that due to its dynamic placement, DUX is 11% faster than the fixed *Config-1* and 6% slower than *Config-2* but utilizes 63% less SSD storage capacity than *Config-2*. Although *Config-2*, the full SSD-based Hadoop configuration provides the fastest solution, DUX offers a practical alternative with fewer number of SSDs. Thus DUX is effective in its goal of achieving higher performance by effectively utilizing the SSD tier.

In summary, evaluation of DUX using a 9-node Hadoop cluster shows an 11% speed-up in application completion times when the SSD tier accounted for only 10% of the total available storage. Our trace-driven simulations using synthetic Facebook workloads show that DUX performs only 5% slower while using 5.5× fewer SSDs when compared to a Hadoop cluster provisioned entirely with SSDs.

6.4 Chapter Summary

In this chapter, we have presented the design and implementation of DUX, an application-attuned dynamic data management system for Hadoop. DUX aims to improve overall I/O throughput of Hadoop via effective use of SSDs as a cache, not for all data, but only for workloads that are expected to benefit from SSDs. The novelty of our approach is that it profiles the Hadoop application performance on SSDs and HDDs, analyzes the I/O behavior, and considers the available SSD capacity to dynamically place data in an appropriate storage tier. We also design placement and retrieval schemes to utilize the caching tier for popular data, and support prefetching of input data, consequently maximizing the impact of SSDs on overall performance. We evaluate our implementation of DUX on a 9-node cluster and show that DUX achieves 11% improvement in application completion times when only 10% of its storage is provided by SSDs.

Chapter 7

A Heterogeneity-Aware Hadoop Workflow Scheduler

The *DSF* hardware 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. The goal of ϕSched 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 [52]. Moreover, the workflows are realized through a variety of high-level tools and languages [86] instead of manual MapReduce programming. Therefore, systems such as Oozie [87], Nova [88], and Hadoop+Kepler [89] 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 [87]. 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.

ϕSched considers the 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[1], 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 [52, 93] 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 [239], 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* [240] 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 [241] 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 lo-

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

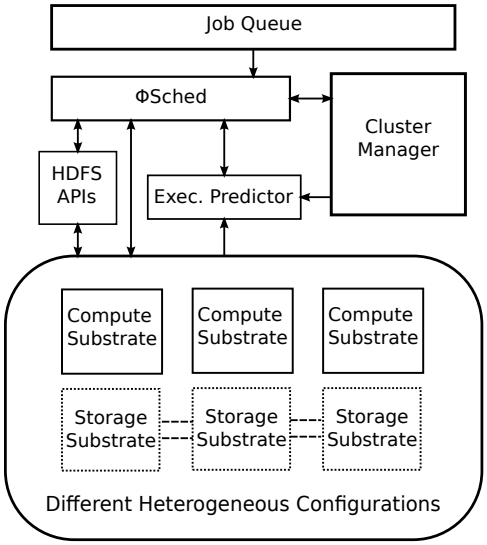


Figure 7.1 ϕSched architecture overview.

cality 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.

7.1 Design of ϕSched

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.

7.1.1 Architecture Overview

Figure 7.1 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.

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.

7.1.2 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 [235], which cover a wide range of workload behavior such as batch processing, iterative jobs and interactive querying. This is motivated by previous research [52, 93] 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 the number of map and reduce tasks therein, the utilization of a single map/reduce task, as well as the input data size. Algorithm 3 shows the steps taken by the Cluster Manager to update the available resources, R_a, at a cluster as jobs are submitted and complete.

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

forall the *job j in job queue* **do**
    forall the *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 4: Steps taken by the Execution Predictor.

7.1.3 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 4 shows the steps taken by the Execution Predictor.

Table 7.1 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.

7.1.4 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 7.1, 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 3 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.

7.1.5 Hardware-Heterogeneity-Aware Scheduling

ϕSched realizes heterogeneity-aware scheduling as follows. 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.

7.2 Implementation of ϕSched

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

Cluster Manager and Execution Predictor We have implemented a proof-of-concept Cluster Manager and Execution Predictor in Python. We used the *SAR* tool [210] 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 [242] (package name *python-mysqldb*) to enable this interaction.

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 7.1. 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.

7.3 Evaluation of ϕSched

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

7.3.1 Experimental Setup

We used the Amazon EC2 [243] to perform our experiments. We used four clusters of eight homogeneous nodes, where each cluster had a different hardware configuration as listed in Table 7.2[2]. 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.

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

Table 7.2 Hardware configurations considered in our experiments.

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 7.3 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

7.3.2 Studied Applications

In this section, we describe 12 applications from the well-known Hadoop HiBench Benchmark Suite [235], which we have used in our study. These applications are representative of batch processing jobs, iterative jobs and interactive querying jobs. Table 7.3 lists the applications, and for each also summarizes parameters such as the input and output data size, and the number of mappers and reducers.

7.3.3 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 7.3. The results discussed below are average of four executions; the standard deviation across the runs was observed to be negligible.

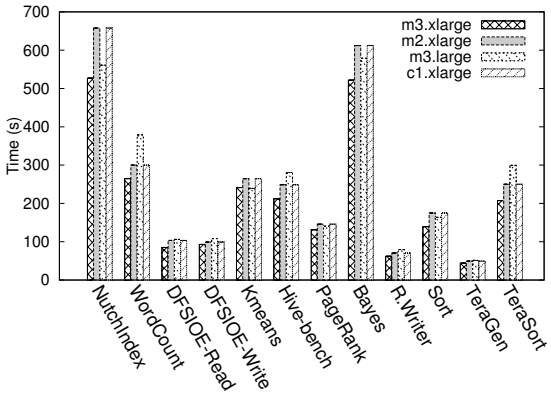


Figure 7.2 Application execution time on the studied hardware configurations.

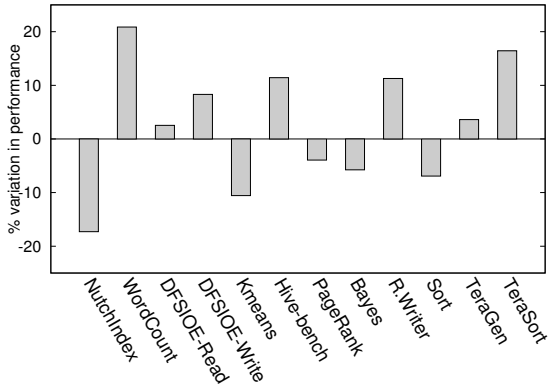


Figure 7.3 Performance improvement observed on *m3.large* compared to *m2.xlarge*.

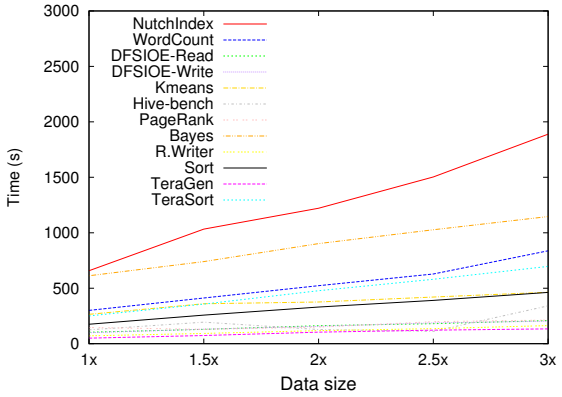


Figure 7.4 Effect of increasing the input data size on execution time.

Performance Comparison In this test, we measured the execution time of our test applications on the studied clusters. As shown in Figure 7.2, 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 *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 7.3 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 intercon-

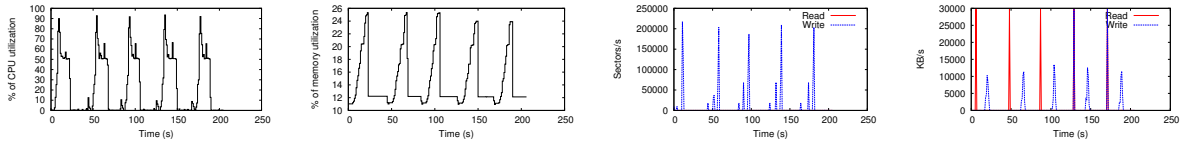
nects. Similar pattern is also observed while comparing the execution time of *m3.xlarge* with *c1.xlarge*, in fact the best case placement will perform 34% faster than the worst case. These results validate our claim that the performance of the application varies significantly across various cluster configurations, and can be problematic if the entire deployment is managed using a single Hadoop instance or in a hardware oblivious manner.

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 7.4 shows the execution time of the applications under varying data size for the *m2.xlarge* configuration. We increase the input size shown in Table 7.3 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.

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*.

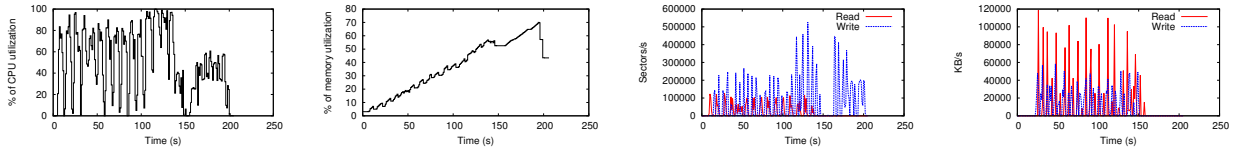
Kmeans is an iterative application and the size of the data does not vary between iterations. Moreover, Figure 7.5 shows that the resource usage is similar across iterations. We observe that *Kmeans* is CPU bound and uses almost 28% of the CPU, on average. Next, Figure 7.6 shows the usage characteristics 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 7.7. We see that *Bayes* has 2× the execution time compared to both *Kmeans* and *TeraSort*. We observe that in spite of the increased



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

Figure 7.5 Resource usage characteristics of *Kmeans* on *m3.xlarge*.



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

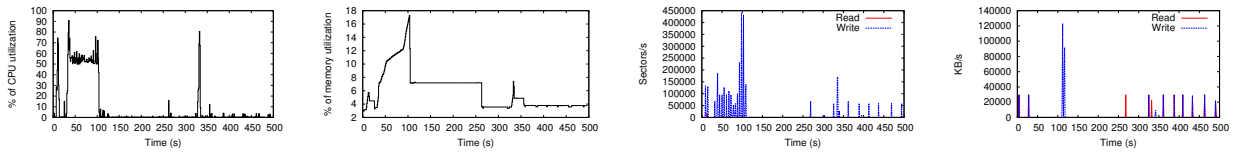
Figure 7.6 Resource usage characteristics of *TeraSort* on *m3.xlarge*.

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.

7.3.4 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.

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 7.8 shows the distribution of blocks, which we determine by parsing HDFS logs. We find that both the default placement and region-aware placement distributes the data uniformly



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

Figure 7.7 Resource usage characteristics of *Bayes* on *m3.xlarge*.

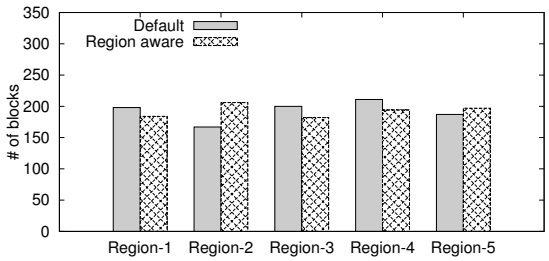


Figure 7.8 Distribution of data blocks across different regions under default and region-aware policies.

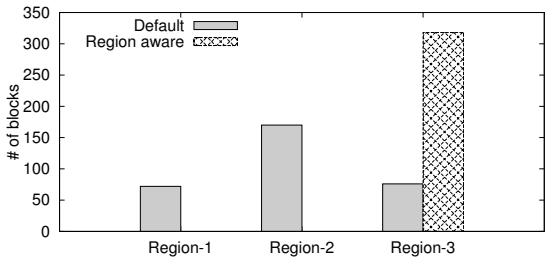


Figure 7.9 Number of blocks that are replicated across multiple regions under default and region-aware policies.

among different regions. However, a closer analysis (Figure 7.9) 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.

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 7.10 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

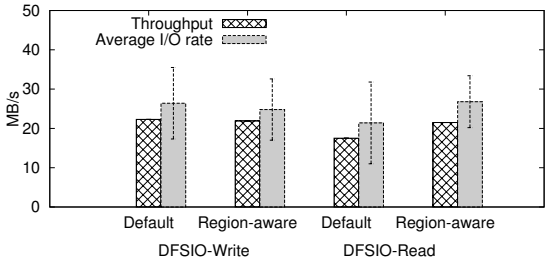


Figure 7.10 Overall write throughput and average I/O rate per map task in *TestDFSIOE-Write* and *TestDFSIOE-Read* under default and region-aware policies.

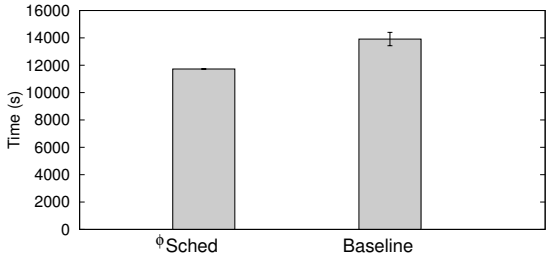


Figure 7.11 Execution time of the test workflow under ϕSched and hardware oblivious workflow scheduler.

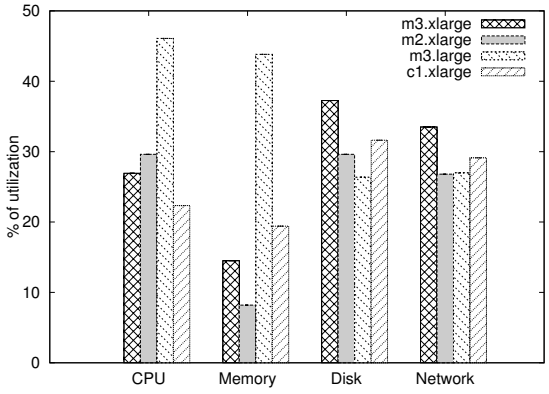


Figure 7.12 Average hardware usage of the test workflow under hardware oblivious scheduler.

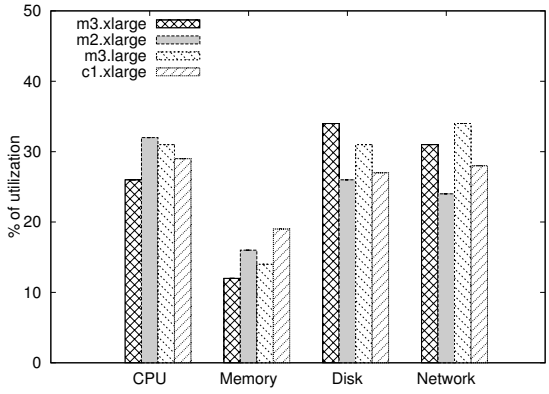


Figure 7.13 Average hardware usage of the test workflow under ϕSched.

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 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.

7.3.5 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 7.2, 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 7.3. 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 7.11. 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 7.12 and 7.13 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× 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× 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.

7.4 Chapter Summary

In this chapter, 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%.

Chapter 8

Towards Energy Awareness in Hadoop

In this work, we propose ϵSched to improve the application-resource match by considering heterogeneous Hadoop deployments that comprise of one or more homogeneous sub-clusters. The set of tasks to be executed on the heterogeneous deployment cluster will be scheduled to the sub-clusters such that the total energy consumption is minimized while the performance goals specified in the Service Level Agreement (SLA) are met. We propose simple, application characteristic-aware task scheduling in Hadoop to reduce the power consumption or to improve the throughput.

8.1 Design of ϵSched

To this end, we present ϵSched, a heterogeneity-aware and power-conserving task scheduler for Hadoop. ϵSched extends our own ϕSched system [94] – a hardware characteristic-aware scheduler that improves the resource-application match. We extend Hadoop's hardware-aware scheduler, which is optimized only for performance, to be an energy efficient scheduler. We adopt a quantitative approach where we first study detailed behavior of applications, such as performance and power characteristics, of various representative Hadoop applications running on four different hardware configurations. Next, we incorporate findings of these experiments into ϕSched. To ensure that job associated data is available locally to (or nearby) a cluster in a multi-cluster deployment, ϕSched configures a single Hadoop Distributed File System (HDFS) [94] instance across all the participating clusters. As part of ϵSched, we also design and implement a region-aware data placement and retrieval policy for HDFS in order to reduce the network overhead and achieve cluster-level data locality.

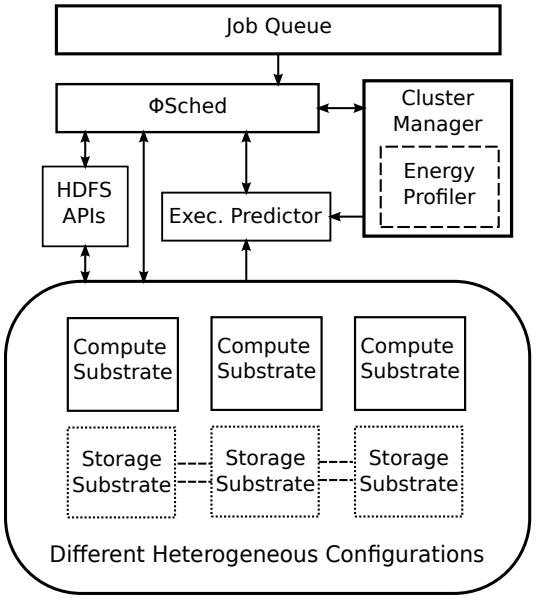


Figure 8.1 ϵSched architecture overview.

8.1.1 Energy Profiler

As shown in Figure 8.1, ϵSched integrates an Energy Profiler component to ϕSched to track, record and analyze the power usage characteristics of the application on a particular hardware. ϵSched computes the performance to power ratio for different jobs in each cluster. As shown in Section 8.2, different cluster show different performance and power characteristics. There is no single cluster that is optimal for power and performance for all workloads.

As illustrated in Algorithm 5, when a job is to be scheduled in one of the homogeneous sub-clusters, $C_1, C_2, ..., C_n$ in a heterogeneous cluster, C, deployment, the Energy Profiler component accesses its profiled power characteristics. The sub-clusters are arranged in the order of performance to power ratio. Energy Profiler will traverse through the ordered list of sub-clusters to find the cluster that would meet the SLA of the workflow to be scheduled. Upon recognizing the optimal sub-cluster that will ensure the least power usage while meeting the SLA requirements, the Cluster Manager in ϕSched checks for the availability of resources in optimal sub-cluster to schedule the workflow W. If the availability of resources is determined, W is scheduled, else the process is repeated to search for the next optimal cluster.

```
For workload W, arrange clusters C_1 to C_n in C in the descending order of
performance to power ratio ;
foreach C_i in C do
    if C_i meets SLA then
        if isResourceAvailable(C_i,W) then
            Schedule W in C_i;
            Increment cluster utilization for C_i;
            break;
        end
    end
end
```

Algorithm 5: Energy Profiler in ϵSched.

8.1.2 Discussion

The HDFS enhancement provided by ϕSched will ensure data availability in a smaller set of nodes by modifying the placement policy. At least one replica of a file is stored in a small subset of nodes (i.e., the sub-cluster with low power-usage) called Covering Subset [244]. This will enable us to apply energy harvesting techniques such as turning a sub-cluster off or running in it low power mode for any of the under-utilized sub-clusters that are not a part of Covering Subset.

It is important to note that the power characteristics of a workload on a cluster is linearly dependent on the data size. Thus the profiler component can linearly extrapolate the power characteristics to different data sizes based on the studied workloads. Similar observations were made for performance characteristics in ϕSched [94]. Moreover the number of sub-clusters in a real deployment will be less than ten, so the overhead constituted by the energy profiler component is minimal.

8.2 Evaluation of ϵSched

We evaluate the energy characteristics of Hadoop applications on using a real deployment on a medium-scale cluster. We first study the characteristics of 8 representative Hadoop applications on three different cluster hardware configurations.

Table 8.1 Hardware configurations considered in our experiments.

Name	CPUs	RAM (GB)	Storage (GB)	Network	Map Slots	Reduce Slots
Cluster-1	16	16	HDD	10 Gbps	8	4
Cluster-2	2	2	HDD	128 Mbps	2	2
Cluster-3	8	8	SSD	1 Gbps	4	2

8.2.1 Experimental Setup

We used three clusters of five homogeneous nodes each, where each cluster has a different hardware configuration as listed in Table 8.1. 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 varying number of map and reduce slots depending on the configuration of the machines (Table 8.1) along with a DataNode component.

For measuring the power usage we used Watts Up? PRO [245] power meters in the worker nodes. The power values represented in this section are usage characteristics of one worker node in every sub-cluster. All worker nodes in a single sub-cluster are homogeneous and showed similar power characteristics. We do not consider the power consumption of the master node, as we do not propose any optimization to the Hadoop master components.

8.2.2 Studied Applications

We have used 8 applications from the well-known Hadoop HiBench Benchmark Suite [235] in our study. These applications are representative of batch processing jobs and iterative jobs. Table 8.2 lists the applications, and summarizes parameters, i.e., the input and the output data sizes, the number of mappers and reducers, for each application.

In this experiment, we measured the execution time of our applications on each of the studied clusters. As shown in Figure 8.2, the execution time of the applications varies across different cluster configurations. We find that across all applications, on average, *Cluster-1* performs 43% and 31% faster than *Cluster-2* and *Cluster-3*, respectively. However, we observe that the variation in performance is not similar across all applications. For instance, in the case of *DFSIOE-Read* which involves significant I/O, *Cluster-3* performs 14% faster than *Cluster-1*, whereas for the same cluster, *PageRank* performs only 22% slower.

Table 8.2 Representative MapReduce (Hadoop) applications used in our study.

Application	Map		Mappers	Reducers
	Input	Output		
WordCount	6 GB	12 KB	120	8
DFSIOE-Read	8 GB	–	128	1
DFSIOE-Write	8 GB	–	128	1
Kmeans	1 GB	1 GB	20	1
PageRank	128 MB	12.5 MB	16	8
Bayes	128 MB	4.5 GB	16	1
Sort	6 GB	3 GB	120	8
TeraSort	15 GB	15 GB	240	8

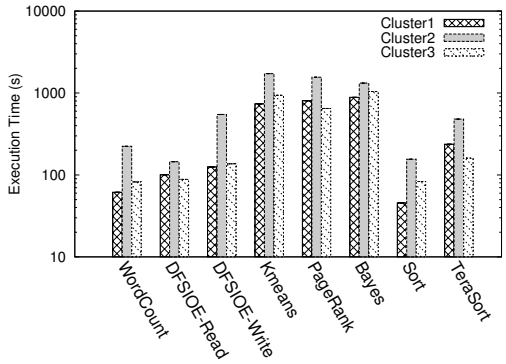


Figure 8.2 Time taken for each application in studied cluster.

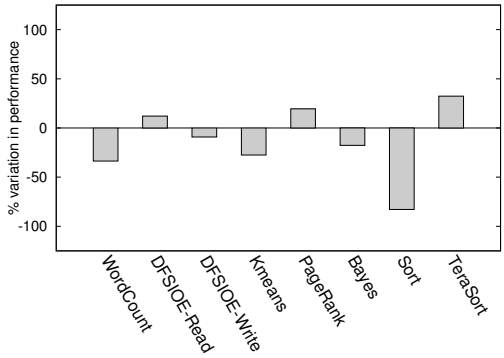


Figure 8.3 Performance improvement observed on *Cluster-1* compared to *Cluster-2*.

To study this variation in detail, we compared the performance under *Cluster-1* and *Cluster-2* across all the studied applications. Figure 8.3 shows the results. For applications such as *WordCount*, *DFSIOE-Write*, *Kmeans*, *Bayes*, and *Sort*, *Cluster-1* performs better, while for the rest of the applications *Cluster-2* performs better. One reason for this is the varying resource requirements of the applications. For example, *Kmeans*, which is a CPU intensive application, performs 27.5% faster in *Cluster-1* that has more memory, and *TeraSort*, which involves significant network and I/O usage, performs better in *Cluster-3* that has better interconnects.

8.2.3 Power Usage Comparison

In this experiment, we measured the total power consumption of our test applications on the studied clusters. As shown in Figure 8.4 (Y-axis — log scale), the total power consumption of the applications varies across different cluster configurations. As expected, the power

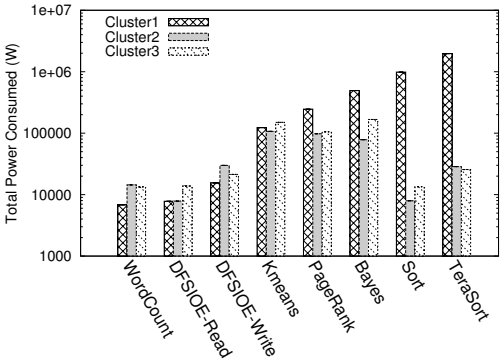


Figure 8.4 Total power consumption for each application in studied cluster.

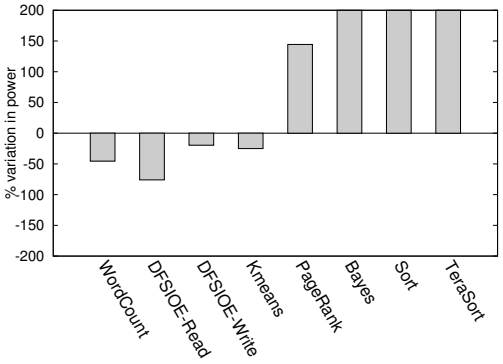


Figure 8.5 Power Consumption observed on *Cluster-1* compared to *Cluster-2*.

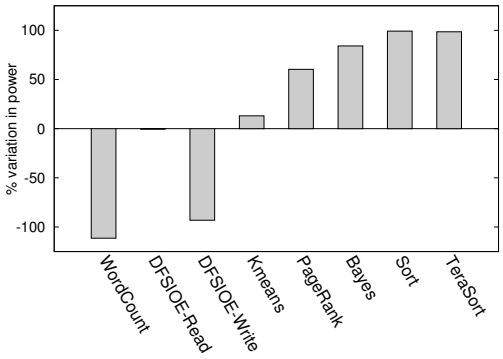


Figure 8.6 Power Consumption observed on *Cluster-1* compared to *Cluster-3*.

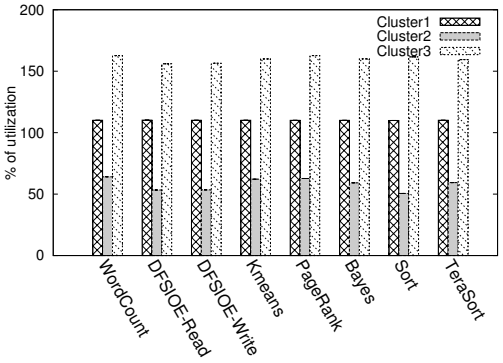


Figure 8.7 Average power consumption for each application in studied cluster.

consumption of an application is not only dependent on the underlying hardware architecture but also on the execution time of the application. For applications such as *WordCount*, *DFSIOE-Write* and *DFSIOE-Write*, the optimal power consumption is observed when executed on *Cluster-1*; for *Kmeans*, *PageRank*, *Bayes* and *Sort* the optimal power consumption is observed under *Cluster-2*; finally, *TeraSort* shows the least power consumption under *Cluster-3*.

To further study this variation in detail, we compared the power consumption under *Cluster-1* and *Cluster-2* across all the studied applications. Figure 8.5 shows the results. For applications such as *WordCount*, *DFSIOE-Read*, and *DFSIOE-Write*, *Cluster-3* shows lower power consumption while for *Kmeans*, *PageRank*, *Bayes*, *Sort* and *TeraSort* *Cluster-1* shows the lower power consumption. Similarly Figure 8.6 shows the comparison of power consumption between *Cluster-1* and *Cluster-3*. For applications such as *WordCount*, *DFSIOE-Write*, *DFSIOE-Write* and *Kmeans*, *Cluster-3* performs better while *Cluster-1* performs better for the rest of the applications.

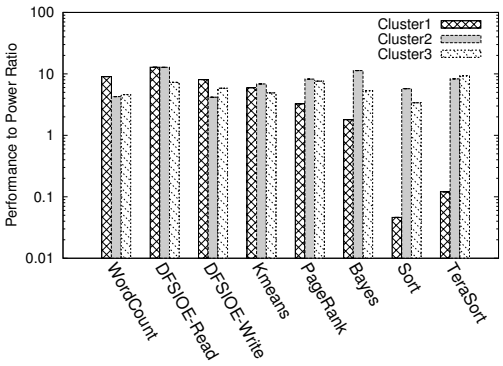


Figure 8.8 Performance to power ratio for each application in studied cluster.

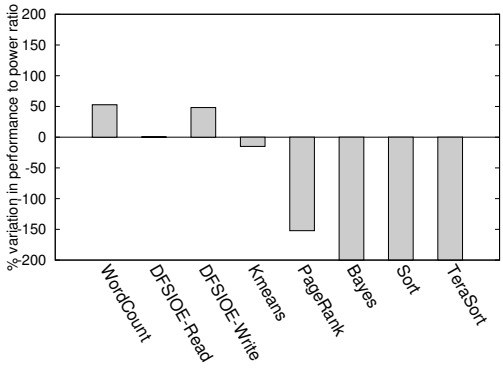


Figure 8.9 Performance to Power ratio improvement observed on *Cluster-1* compared to *Cluster-2*.

We observe that in spite of the high execution time, for a subset of applications such as *WordCount*, *DFSIOE-Read* and *DFSIOE-write*, the total power consumption is the least in *Cluster-2* because of its low average power consumption as shown in Figure 8.7. It is observed that *Cluster-2* consumes the least power and *Cluster-1* and *Cluster-3* consume $2\times$ and $3\times$ the power respectively. A similar trend is observed across all applications.

8.2.4 Performance to Power Ratio Comparison

Figure 8.8 compares the ratio (higher the better) of performance to the total power in order to find the optimum cluster in terms of both performance and power. For applications such as *WordCount*, *DFSIOE-Write* and *DFSIOE-Write*, the optimal cluster is *Cluster-1*, for *Kmeans*, *PageRank*, *Bayes* and *Sort* the optimal cluster is *Cluster-2*, and for *TeraSort* the optimal cluster is Cluster-3. In Figure 8.9, detailed examination reveals a variation in the application behavior, similar to the above cases.

For our next set of experiments, we develop an accurate simulator for ϵSched to observe the power consumption and the execution time of the considered clusters. Our fine-grained simulator takes into account details such as the effect of compute capacity, network and storage infrastructure, and application-hardware affinity with reference to power and execution time. We simulate a 300-node cluster, consisting of three 100-node homogeneous sub-clusters. The configuration of each node in a sub-cluster is similar to *Cluster-1*, *Cluster-2* and *Cluster-3* shown in Table 8.1. We use the publicly available synthetic Facebook production traces [211] for driving the simulation. We replay a snippet of these traces using HiBench [212] applications (Table 8.2). The traces run for a length of 3 hours under baseline Hadoop (hardware

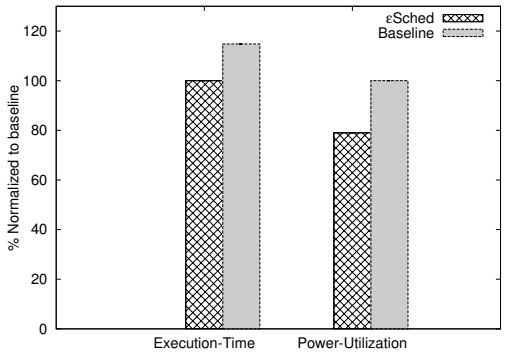


Figure 8.10 Power and performance improvement of ϵSched over Hadoop.

oblivious scheduling).

Figure 8.10 shows the completion time and the power consumed by the workloads under both ϵSched and hardware oblivious scheduling. To highlight the power savings achieved using ϵSched we neglect the power consumption of the system in idle state, assuming that power management schemes can be applied to nodes that are in idle state for a longer period of time. ϵSched achieves both performance benefits and power savings over baseline Hadoop. By the application-hardware match improvements in ϵSched, the completion time of the application improves by 12.8%, while consuming 21% less power than baseline Hadoop.

In summary, the above experiments validate the claim that different application-hardware interactions produce different power consumption and performance characteristics.

8.3 Chapter Summary

In this paper, we design and implement ϵSched, a novel hardware-aware workflow scheduler for Hadoop. We observe that different workflows have different performance and power usage characteristics under varying cluster configurations, and making workflow managers aware of the underlying configuration can significantly increase overall performance and power consumption. We study the impact of ϵSched on Hadoop application performance using a range of representative applications and configuration parameters. Our evaluation shows that ϵSched managing three different clusters can achieve performance improvement of 12.8%, on average, while consuming 21% less power when compared to hardware oblivious scheduling.

Chapter 9

Conclusion

The dissertation presents the design of a resource management framework for heterogeneous DSFs. A major problem faced in evolving DSFs is to efficiently handle increasing heterogeneity in the underlying hardware infrastructure. The infrastructure that supports DSFs is becoming increasingly heterogeneous. DSF computation is tending towards low-cost, power-efficient clusters that employ traditional servers along with specialized resources and high-throughput DSF storage is trending towards hybrid and tiered approaches that use large in-memory buffers, SSDs, etc., in addition to traditional disks. One reason for this is that different types of hardware such as CPUs, memory, storage, and network are deployed when large clusters typically go trough upgrade phases. However, a more crucial driver for the heterogeneity is the rise of hybrid systems. The net effect of the above trends is that DSF cluster nodes exhibit orders of magnitude variation in terms of compute power, cluster integration and programmability. While recent studies have shown that use of specialized accelerators for Hadoop is desirable, sustaining DSFs on such resources is challenging. This is because most modern DSFs are designed to run on homogeneous clusters and cannot effectively handle general purpose workloads on heterogeneous resources.

In this dissertation, we research to design, implement, and evaluate an application-characteristics-aware resource manager for DSFs, which adopt a quantitative approach where we first study detailed behavior of various DFS applications running on different hardware configurations and propose application-attuned dynamic system management in order to improve the resourceapplication match. LSN presents a novel enhancement for Hadoop, which divides a traditional Hadoop rack into several sub-racks, and consolidates the disks attached to each of the sub-racks compute nodes into a shared LSN for servicing the sub-rack. The scope of a single LSN can range from serving a few nodes to perhaps a complete Hadoop cluster rack depending on workload and usage characteristics. The thesis also explores the utility of heterogeneous storage devices in Hadoop and address challenges therein

by designing hat$S$, a heterogeneity-aware tiered storage for Hadoop. hat$S$ logically groups all storage devices of the same type across the nodes into an associated "tier." hat$S$ also provides custom APIs for seamless data transfer across the tiers and management of stored data in each tier. These features allow hat$S$ to integrate heterogeneous storage devices into Hadoop to extract high I/O performance. AptStore analyzes the I/O access patterns and suggests storage policies to increase the overall read throughput and the storage efficiency of the system. To tackle compute heterogeneity, 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. DUX provides an application-attuned storage management for Hadoop, it predict the impact of the I/O accesses on execution time and choose an appropriate tier for storing the intermediate data and for the jobs waiting in the job queue, prefetch the input data into the SSD tier if it has not been selected by AptStore. Derby*hat*S provides a holistic approach by combining the above component and enabling a workflow scheduler, constituted of two components. ϕSched manages the compute heterogeneity and DUX coupled with AptStore manages the storage the storage substrate to exploit heterogeneity.

9.1 Future Research

This is a DSF era in which heterogeneous-aware resource management techniques serve as a fundamental enabling technology. In this dissertation, we have addressed the challenges of managing resources and enhanced heterogeneous storage management for emerging DSFs. Nevertheless, there exists a number of open questions related to the efficient use of computing resources in the cloud. In the following, I outline my vision that are natural extensions of the techniques discussed in this dissertation especially in data analytics performance improvement, areas of resource management and storage optimization in DSFs.

9.1.1 Evaluating the Impact of Heterogeneous DSF Design Changes in Simulation Frameworks

DSFs, both native and in the cloud, are now integral in delivering fast time-to-solution in a myriad of fields. Simulation-based modeling to re-architecture DSFs can have a profound

impact on how we think of developing, deploying, and using the next-generation heterogeneous DSF applications and systems. Evaluating the impact of DSF design changes in real clusters is cumbersome, consider the following steps in determining whether a new network topology for DSF would be beneficial. First, the new cluster networking must be configured, typically requiring labor-intensive rewiring, and designing new routing. Some cluster nodes may also need to be repositioned, e.g., to make room for the new network components, further requiring reconfiguring of support infrastructure, e.g., electrical sockets, cooling, etc. Second, a set of representative benchmarks must be run to test cluster performance. Third, the results have to be assembled and analyzed. Then, the whole process has to be repeated for varying cluster sizes to ensure that the observed behavior is not merely an artifact of a particular cluster/benchmark instance. Consequently, testing new DSF designs on a real cluster quickly becomes overwhelming, forcing many DSF cluster designers to simply use generic models that yield sub-optimal realizations. On the other hand, using integrated simulations can simplify, speedup, and automate explorations in the DSF design space.

To this end, simulations can enable cost-effective exploration of new hardware and software concepts. A benefit of simulations is that they enable investigation of innovative designs in DSF, even if they are merely conceptual and have not been completely implemented, e.g., novel networking hardware. Such exploratory studies are not cost-effective or even possible if done on a real cluster. One example is the study of different storage-compute configurations for scalability. For example, Network Attached Storage (NAS) can be employed in DSF if the loss of data locality is amortized using an innovative design, e.g., via using novel interconnects. Another example is to revisit optimizations, such as anticipatory scheduling and history based prediction, in the context of DSF All of the above mentioned design explorations become even more difficult for virtualized DSF, as now one has to also consider the interactions of DSF with cloud VM management, as well as virtualized topology. Simulations are ideal here, as they can model entire systems and provide insights into the relative performance of different designs.

Simulations can predict DSF behavior online to drive better resource management simulations to provide means for quickly estimating system behavior. We can capture the current state of a live DSF system, simulate, and predict the expected interactions for the near future. This information can then be used to guide DSF resource management. Thus, our proposed simulation framework can also be used as a scheduling and resource management tool for DSF. Developing such an encompassing DSF simulation framework is a daunting task. Of particular importance is extending and verifying the simulator at scale, ensuring

its modularity, and developing associated flexible and expandable APIs. Another issue is to accurately capture DSF behavior, especially given the myriad of design parameters, as well as validation of the sub-components and the system as a whole. Finally, the overhead of running the simulation online within DSF should be kept to a minimum.

9.1.2 Application Behavior Analyzer and Optimizer

A major problem faced in evolving DSFs applications is that they typically are complex workflows comprising multiple different kernels. The kernels can be diverse, e.g., compute-intensive processing followed by data-intensive visualization, and thus preclude the use of extant static global optimizations in DSFs. Typical modern data analysis application workflows comprise of multiple independent tasks exhibiting different characteristics during application execution, and the underlying infrastructure is also becoming heterogeneous. Current optimizations in compilers and runtime systems are severely limited in handling user defined functions (UDFs) , such as the ones implementing custom mappers, reducers, and mergers. UDFs currently are treated as black boxes, whose properties and potential for parallelization on different types of hardware remain unexplored. These black-box UDFs are increasingly composed into complex dataflows, but the runtime system remains unaware of their essential characteristics, and as a result, opportunities for many cross-task and cross-job optimization opportunities are lost. There is an urgent need for an automated, cross-layer performance optimization framework for DSFs, which encompass new compile- and run-time optimizations, and also supports their seamless collaboration.

Many DSFs are now running complex workflow applications with user defined functions, which are currently treated as black boxes. One way to improve performance is to expose the heterogeneity and allow programmers to handle it explicitly. However, this would be counterproductive and break the effective programming model that makes modern DSFs easy-to-use. Rather an indepth understanding of properties of complex workflows is needed to do away with the unsustainable black box model and to improve the DSF cluster design and resource management strategies. An application-aware optimizer can exploit a wide spectrum of UDF properties such as data partitioning and functional/algebraic properties, such as monotonicity and commutativity. For example, if the input dataset is sorted, and a transfer function is monotonically increasing, then the output is guaranteed to be sorted as well. If an intelligent DSF runtime system can infer this property, it can eliminate a subsequent network hungry data-shuffling phase, therefore significantly improving the runtime

performance. Similarly, an optimizer in the workflow execution engine can move a cheap filter ahead of a more expensive operation with which the filter commutes. However, if the operation involves a UDF, which is treated as a black box by todays runtime systems, it would be impossible for the optimizer to decide whether the two operations commute.

To support writing of efficient data parallel programs without sacrificing programmability, we need to hide the details of the DSF runtime and leverage automated optimizations aggressively. Such optimizations may be used at compile time to remove redundant operations from the dataflow, and at run time to improve task scheduling and data management. However, traditional optimizations in compilers and runtime sysems are severely limited in dealing with UDFs. UDFs are the programmer provided codes for implementing custom operations, such as mappers, reducers, and mergers in DSF. Unlike standard operators with known cost and data reduction properties, UDFs are black boxes. Since neither the dataflow optimizer nor the runtime system is aware of the essential characteristics of these UDFs, many cross-task and cross-job optimization opportunities are lost. As a result, todays data parallel programs have to rely on hand optimization, which is ineffective, and limits the programmers productivity and code re-use.

The findings of my thesis are applicable to other research areas as well. Datacenters infrastructure is becoming increasingly heterogeneous. A more crucial driver for the heterogeneity is the rise of hybrid systems. The compute nodes in modern large-scale distributed systems often boasts of multicore processors with tightly coupled accelerators. Numerous current products from major vendors package a few general-purpose cores (e.g., x86, PowerPC) and several accelerators (e.g., SIMD processors, GPUs, FPGAs), yielding power-efficient and low-cost compute nodes with performance exceeding 100 Gflops per chip. Therefore, specialized embedded devices are also gaining popularity in data centers. Similarly, storage systems are increasingly employing hybrid and heterogeneous storage devices such as Solid State Disks (SSD), Ram Disks and Network Attached Storage (NAS) to yield very high I/O rates at acceptable costs. As a result, a clear disconnect between the design and actual implementations arises, which causes breakdown of design-time assumptions, consequently leading to lost optimization opportunities, degraded performance, and increased failures. There is a crucial need for a resource managers that can capture the true characteristics of the emerging heterogeneous datacenters and help robust development of the systems and application software stacks. The findings of my thesis are applicable for a broad range of traditional computer science research such as High Performance Computing (HPC) and Operating System. HPC can take advantage of the heterogeneous workflow scheduler and effective way

of employing hybrid and heterogeneous storage devices. Similarly, traditional OS cache can take advantage of the proposed SSD caching techniques for permanent and temporary data.

Bibliography

[1] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1 –11, nov. 2011.

[2] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[3] Michael A. Kozuch, Michael P. Ryan, Richard Gass, Steven W. Schlosser, David O'Hallaron, James Cipar, Elie Krevat, Julio López, Michael Stroucken, and Gregory R. Ganger. Tashi: location-aware cluster management. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 43–48, New York, NY, USA, 2009. ACM.

[4] Kang Jaewon, Zhang Yanyong, and B. Nath. Tara: Topology-aware resource adaptation to alleviate congestion in sensor networks. *Parallel and Distributed Systems, IEEE Transactions on*, 18(7):919 –931, july 2007.

[5] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[6] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2).* Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008.

[7] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2).* Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008.

[8] Thilina Gunarathne. Commercial iaas/paas ii: Windows azure and twister4azure. *Science Cloud Summer School 2012*, 07/2012 2012.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[10] Apache Software Foundation. Hadoop, 2015. `http://hadoop.apache.org/`.

[11] Vinod Kumar Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. ACM SOCC*, 2013.

[12] Baldeschwieler, Eric. Hortonworks Manifesto.

[13] Dhruba et.al Borthakur. Apache hadoop goes realtime at Facebook. Proc. ACM SIGMOD, 2011.

[14] Amazon. Amazon Elastic MapReduce (EMR).

[15] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[16] Apache. Apache Mahout. `http://mahout.apache.org/`.

[17] Apache Avro, 2011.

[18] Ariel Rabkin and Randy Katz. Chukwa: a system for reliable large-scale log collection. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[19] Stephen Baker. How the new york times uses clouds, 2007. http://www.businessweek.com/the_thread/blogspotting/archives/2007/12/how_the_new_yor.html.

[20] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[21] Jacek Cala and Paul Watson. Automatic software deployment in the azure cloud. In Frank Eliassen and Rdiger Kapitza, editors, *Distributed Applications and Interoperable Systems*, volume 6115 of *Lecture Notes in Computer Science*, pages 155–168. Springer Berlin / Heidelberg, 2010.

[22] The rackspace cloud. http://www.rackspacecloud.com/.

[23] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, pages 261–272, 2011.

[24] Herodotos Herodotou. Hadoop Performance Models. Technical Report CS-2011-05, Duke University, February 2011.

[25] Apache. Apache Pig. `http://pig.apache.org/`.

[26] Apache. Apache HBase. `http://hbase.apache.org/`.

[27] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive- a warehousing solution over a map-reduce framework. In *IN VLDB '09: PROCEEDINGS OF THE VLDB ENDOWMENT*, pages 1626–1629, 2009.

[28] Grzegorz Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, pages 135–146, New York, New York, USA, 2010. ACM.

[29] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[31] Pivotal. Analytics workbench, 2010. `http://www.ndm.net/datawarehouse/Greenplum/greenplum-`

[32] Jiong Xie et al. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Proc. IEEE IPDPSW*, 2010.

[33] Hannes Payer et al. Combo drive: Optimizing cost and performance in a heterogeneous storage device. In *WISH*, 2009.

[34] Feng Chen, David A Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proc. ACM SC*, 2011.

[35] Li-Pin Chang. Hybrid solid-state disks: combining heterogeneous nand flash in large ssds. In *Proc. IEEE ASPDAC Asia and South Pacific*, 2008.

[36] Jeff Dean. Designs, lessons and advice from building large distributed systems, 2012. `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`.

[37] An-I Wang et al. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *Proc. USENIX ATC*, 2002.

[38] Seok-Hoon Kang, Dong-Hyun Koo, Woon-Hak Kang, and Sang-Won Lee. A case for flash memory ssd in hadoop applications. In *IJCA*, volume 6, 2013.

[39] Karthik Kambatla and Yanpei Chen. The truth about mapreduce performance on ssds. In *Proc. USENIX LISA*, 2014.

[40] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: a facebook messages case study. In *Proc. FAST*, 2014.

[41] Sangwhan Moon, Jaehwan Lee, and Yang-suk Kee. Introducing ssds to the hadoop mapreduce framework. In *Proc. IEEE cloud*, 2014.

[42] R.T. Kaushik, M. Bhandarkar, and K. Nahrstedt. Evaluation and analysis of greenhdfs: A self-adaptive, energy-conserving variant of the hadoop distributed file system. In *Proc. IEEE CloudCom*, 2010.

[43] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proc. ACM EuroSys*, 2011.

[44] How Clean is Your Cloud?, 2012. http://www.greenpeace.org/international/en/publications/Campaign-reports/Climate-Reports/How-Clean-is-Your-Cloud/.

[45] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *ACM SIGOPS 22nd*, 2009.

[46] David G Andersen and Steven Swanson. Rethinking flash in the data center. *IEEE micro*, 2010.

[47] Byung-Gon Chun, Gianluca Iannaccone, Giuseppe Iannaccone, Randy Katz, Gunho Lee, and Luca Niccolini. An energy case for hybrid datacenters. *ACM SIGOPS*, 2010.

[48] Cathy May, Ed Silha, Rick Simpson, Hank Warren, et al. *The PowerPC Architecture: A specification for a new family of RISC processors*. Morgan Kaufmann Publishers Inc., 1994.

[49] Jim Turley. Coldfire doubles performance with v4. *Microprocessor Report*, 26, 1998.

[50] John R Hauser and John Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. In *IEEE FCCM*, 1997.

[51] Dave Jaggar. *ARM Architecture Reference Manual*. Prentice Hall, 1996.

[52] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endowment*, 5(12):1802–1813, 2012.

[53] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 93–102. ACM, 2010.

[54] Todd C Scofield, Jeffrey A Delmerico, Vipin Chaudhary, and Geno Valente. Xtremedata dbx: an fpga-based data warehouse appliance. *Computing in Science & Engineering*, 12(4):66–73, 2010.

[55] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[56] Jeff A. Stuart and John D. Owens. Multi-gpu mapreduce on gpu clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1068–1079, Washington, DC, USA, 2011. IEEE Computer Society.

[57] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, 2008.

[58] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and HASH Scale. Altivec extension to powerpc accelerates media processing. *Micro, IEEE*, 20(2):85–95, 2000.

[59] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. Mips: A microprocessor architecture. *ACM SIGMICRO Newsletter*, 13(4):17–22, 1982.

[60] Rui Wang and Shiyuan Yang. The design of a rapid prototype platform for arm based embedded system. *Consumer Electronics, IEEE Transactions on*, 50(2):746–751, 2004.

[61] F Tso, David R White, Simon Jouet, Jeremy Singer, and D Pezaros. The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. In *International Workshop on Resource Management of Cloud Computing (to appear, 2013)*, 2013.

[62] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, volume 7, pages 79–87. Citeseer, 2007.

[63] YoungHoon Jung, Richard Neill, and Luca P Carloni. A broadband embedded computing system for mapreduce utilizing hadoop. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 1–9. IEEE, 2012.

[64] Navid Golpayegani and Milton Halem. Cloud computing for satellite data processing on high end compute clusters. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 88–92. IEEE, 2009.

[65] Ming-hsien Yang. High-performance heterogeneous hadoop architecture. 2012.

[66] FADI THABTAH and SUHEL HAMMOUD. Mr-arm: A map-reduce association rule mining framework. *Parallel Processing Letters*, 23(03), 2013.

[67] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza. Mixapart: decoupled analytics for shared storage systems. *USENIX HotStorage*, 2012.

[68] Nusrat S Islam, MW Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 35. IEEE Computer Society Press, 2012.

[69] Yifeng Luo, Siqiang Luo, Jihong Guan, and Shuigeng Zhou. A ramcloud storage system based on hdfs: Architecture, implementation and evaluation. *Journal of Systems and Software*, 2012.

[70] Devesh Tiwari, Simona Boboila, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[71] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, pages 101–114, 2010.

[72] ornl. Top500 super computers. `http://www.top500.org/list/2012/11/`.

[73] Petascale Computing on Jaguar. `http://www.nccs.gov/jaguar/`, 2007.

[74] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, June 2005.

[75] M. Hill and M. Marty. Amdahl's Law in the Multi-core Era. Technical Report 1593, Department of Computer Sciences, University of Wisconsin-Madison, March 2007.

[76] M. Pericàs, A. Cristal, F. Cazorla, R. González, D. Jiménez, and M. Valero. A Flexible Heterogeneous Multi-core Architecture. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–24, September 2007.

[77] Kumar R., K. Farkas, N. Jouppi, P. Ranganathan, and D. M. Tullsen. Processor Power Reduction via Single-ISA Heterogeneous Multi-core Architectures. *Computer Architecture Letters*, 2, 2003.

[78] Kumar R., D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[79] H. Wong, A. Bracy, E. Schuchman, T. Aamodt, J. Collins, P. Wang, G. Chinya, A.Khandelwal Groen, H. Jiang, and H. Wang. Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *Proc. of the 17th IEEE International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Canada, October 2008.

[80] AMD. The Industry-Changing Impact of Accelerated Computing. 2008.

[81] Simon J Cox, James T Cox, Richard P Boardman, Steven J Johnston, Mark Scott, and Neil S OBrien. Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing*, pages 1–10, 2013.

[82] Xiaoyan Gu, Rui Hou, Ke Zhang, Lixin Zhang, and Weiping Wang. Application-driven energy-efficient architecture explorations for big data. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, pages 34–40. ACM, 2011.

[83] Hyeong S Kim, Dong In Shin, Young Jin Yu, Hyeonsang Eom, and Heon Y Yeom. Towards energy proportional cloud for data processing frameworks. In *Proceedings of the First USENIX conference on Sustainable information technology*, pages 4–4. USENIX Association, 2010.

[84] Dhruba Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[85] Apache Oozie. Workflow scheduler for hadoop, 2012.

[86] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Proc. CIDR*, 2011.

[87] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proc. ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 2012.

[88] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki BN Rao, Vijayanand Sankarasubramanian, Siddharth Seth, et al. Nova: continuous pig/hadoop workflows. In *Proc. ACM SIGMOD*, 2011.

[89] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. Kepler + hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. Proc. WORKS, 2009.

[90] K Shvachko, H Kuang, S Radia, and R Chansler. The hadoop distributed file system. In *26th IEEE Symposium on Massive Storage Systems and Technologiese*, 2010.

[91] Hadoop and Disparate Data Stores, 2012. http://blog.gopivotal.com/products/hadoop-and-disparate-data-stores.

[92] Sameer Tiwari. Managing Hot and Cold Data Using a Unified Storage System, 2012. http://blog.gopivotal.com/products/managing-hot-and-cold-data-using-a-unified-storage-system.

[93] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza. Mixapart: Decoupled analytics for shared storage systems. In *Proc. USENIX HotStorage*, 2012.

[94] K.R. Krish, Ali Anwar, and Ali R. Butt. sched: A heterogeneity-aware hadoop workflow scheduler. 2014.

[95] Dhruba Borthakur. Facebook has the world's largest hadoop cluster!, 2010. http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html.

[96] Tom White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[97] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proc. USENIX NSDI*, 2012.

[98] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. USENIX OSDI*, 2008.

[99] Steven Y Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On availability of intermediate data in cloud computations. In *HotOS*, 2009.

[100] George Porter. Decoupling storage and computation in hadoop with superdatanodes. *SIGOPS Oper. Syst. Rev.*, 44:41–46, April 2010.

[101] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. Tritonsort: a balanced large-scale sorting system. In *USENIX NSDI*, 2011.

[102] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proc. USENIX HotOS*, 2011.

[103] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high i/o data rates. *Computing Systems*, 4(4):405–436, 1991.

[104] John H Hartman and John K Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[105] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. USENIX FAST*, 2002.

[106] Brent Welch et al. Scalable performance of the panasas parallel file system. In *Proc. USENIX FAST*, 2008.

[107] Krishnaraj Ravindranathan, Aleksandr Khasymski, Ali Butt, Sameer Tiwari and Milind Bhandarkar. Aptstore:dynamic storage management for hadoop. In *IEEE CloudCom*, 2013.

[108] Edmund B. Nightingale et al. Flat datacenter storage. In *Proc. USENIX OSDI*, 2012.

[109] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: exploiting in-network aggregation for big data applications. In *Proc. USENIX NSDI*, 2012.

[110] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza. Mixapart: decoupled analytics for shared storage systems. *USENIX HotStorage*, 2012.

[111] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SOSP*, 2003.

[112] Bin Fan and et. al. Diskreduce: Raid for data-intensive scalable computing. In *SOCC*, 2010.

[113] Rini Kaushik and Milind Bhandarkar. GreenHDFS: towards an energy-conserving storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[114] W Lang and J. M. Patel. Energy management for mapreduce clusters. In *Proc. VLDB Endowment*, volume 3, pages 129–139. VLDB Endowment, 2010.

[115] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. In *ACM SIGOPS Operating Systems Review*, 2010.

[116] Hrishikesh Amur, James Cipar, and Varun Gupta. The case for evaluating mapreduce performance using workload suites. In *SOCC*, 2010.

[117] Hubert Kario. Lvmts, 2012. `https://github.com/tomato42/lvmts`.

[118] Xiaonan Zhao, Zhanhuai Li, Xiao Zhang, and Leijie Zeng. Block-level data migration in tiered storage system. In *Proc. IEEE ICCNT*, 2010.

[119] Data storage on a multi-tiered disk system, August 2 2005. US Patent 6,925,529.

[120] Susan Feldman and Richard L Villars. The information lifecycle management imperative. *IDC White Paper*, 2006.

[121] Michael Peterson. Ilm and tiered storage. *Storage Networking Industry Association*, 2006.

[122] HDFS-2832. Enable support for heterogeneous storages in hdfs, 2012. `https://issues.apache.org/jira/browse/HDFS-2832`.

[123] Anurag Awasthi, Avani Nandini, Arnab Bhattacharya, and Priya Sehgal. Hybrid hbase: Leveraging flash ssds to improve cost per throughput of hbase. 2012.

[124] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proc. USENIX HotCloud*, 2010.

[125] Paolo Costa et al. Camdoop: Exploiting in-network aggregation for big data applications. In *Proc. USENIX NSDI*, 2012.

[126] Sriram Rao, Benjamin Reed, Joydeep Sarma Sen, Khaled Elmeleegy, and Adam Silberstein. Hotrod: Managing grid storage with on-demand replication. Technical Report YL-2012-004, Yahoo, Apr 2012.

[127] Trushkowsky Beth and et. al. The scads director: scaling a distributed storage system under stringent performance requirements. In *USENIX FAST*, 2011.

[128] Asahara Nakadai and Araki. Load atomizer: A locality and i/o load aware task scheduler for mapreduce. In *IEEE CloudCom*, 2012.

[129] Fangpeng Dong and Selim G Akl. Pfas: a resource-performance-fluctuation-aware workflow scheduling algorithm for grid computing. In *Proc. IEEE IPDPS*, 2007.

[130] James Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task scheduling strategies for workflow-based applications in grids. In *Proc. IEEE CCGrid*, 2005.

[131] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.

[132] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Proc. IEEE IPDPSW*, 2010.

[133] Yuanquan Fan, Weiguo Wu, Haijun Cao, Huo Zhu, Xu Zhao, and Wei Wei. A heterogeneity-aware data distribution and rebalance method in hadoop cluster. In *ChinaGrid Annual Conference (ChinaGrid), 2012 Seventh*, pages 176–181. IEEE, 2012.

[134] Yen-Liang Su, Po-Cheng Chen, Jyh-Biau Chang, and Ce-Kuen Shieh. Variable-sized map and locality-aware reduce on public-resource grids. *Future Generation Computer Systems*, 27(6):843–849, 2011.

[135] Brian Tierney, William Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 260–267. IEEE, 1998.

[136] Thilo Kielmann, Rutger FH Hofman, Henri E Bal, Aske Plaat, and Raoul AF Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. In *ACM Sigplan Notices*, volume 34, pages 131–140. ACM, 1999.

[137] Lei Lu, Hui Zhang, Guofei Jiang, Haifeng Chen, Kenji Yoshihira, and Evgenia Smirni. Untangling mixed information to calibrate resource utilization in virtual machines. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 151–160. ACM, 2011.

[138] HT Kung, Chit-Kwan Lin, and Dario Vlah. Cloudsense: continuous fine-grain cloud monitoring with compressive sensing. *USENIX Hot-Cloud*, 2011.

[139] Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pages 1–8. IEEE, 2012.

[140] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM, 2003.

[141] Andrew Turner, Akkarit Sangpetch, and Hyong S Kim. Empirical virtual machine models for performance guarantees. In *Proceedings of the 24th international conference on Large installation system administration*, pages 1–15. USENIX Association, 2010.

[142] Jin Shao, Hao Wei, Qianxiang Wang, and Hong Mei. A runtime model based monitoring approach for cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 313–320. IEEE, 2010.

[143] Khalid Alhamazani, Rajiv Ranjan, Fethi Rabhi, Lizhe Wang, and Karan Mitra. Cloud monitoring for optimizing the qos of hosted applications. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 765–770. IEEE, 2012.

[144] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, 2009.

[145] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, volume 12, 2012.

[146] Jerome Boulon, Andy Konwinski, Runping Qi, Ariel Rabkin, Eric Yang, and Mac Yang. Chukwa, a large-scale monitoring system. In *Proceedings of CCA*, volume 8, 2008.

[147] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM, 2006.

[148] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience (CCPE)*, 14(13-15):1175–1220, November 2002.

[149] Junwei Cao, Daniel Spooner, James D. Turner, Stephen Jarvis, Darren J. Kerbyson, Subhash Saini, and Graham Nudd. Agent-based resource management for grid computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pages 350–, Washington, DC, USA, 2002. IEEE Computer Society.

[150] Bojan Zagrovic, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. In simulation of folding of a small alpha-helical protein in atomistic detail using worldwide distributed computing. *J. Mol. Biol.*, 2002.

[151] Abigail Morrison, Carsten Mehring, Theo Geisel, Ad Aertsen, and Markus Diesmann. Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Computation*, 17(8):1776–1801, 2005.

[152] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling distributed applications: the simgrid simulation framework. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03, pages 138–, Washington, DC, USA, 2003. IEEE Computer Society.

[153] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[154] Suhel Hammoud. MRSim: A discrete event based MapReduce simulator. In *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, pages 2993–2997. IEEE, August 2010.

[155] Apache Software Foundation. Mumak: Map-Reduce Simulator. `https://issues.apache.org/jira/browse/MAPREDUCE-728`, July 2009.

[156] Kelvin Cardona, Jimmy Secretan, Michael Georgiopoulos, and Georgios Anagnostopoulos. A Grid Based System for Data Mining Using MapReduce. Technical Report TR-2007-02, The AMALTHEA REU Program, 2007.

[157] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[158] Chen Zhang and Hans De Sterck. Cloudwf: A computational workflow system for clouds based on hadoop. In *Cloud Computing*, pages 393–404. Springer, 2009.

[159] Qichang Chen, Liqiang Wang, and Zongbo Shang. Mrgis: A mapreduce-enabled high performance workflow system for gis. In *Proc. IEEE International Conference on eScience*, 2008.

[160] Facebook Corona. Facebook, 2012. https://gigaom.com/2012/11/08/facebook- open-sources-corona-a-better-way-to-do-webscale-hadoop.

[161] CK Wensel. Cascading: Defining and executing complex and fault tolerant data processing workflows on a hadoop cluster, 2008.

[162] David J DeWitt, Erik Paulson, Eric Robinson, Jeffrey Naughton, Joshua Royalty, Srinath Shankar, and Andrew Krioukov. Clustera: an integrated computation and data management system. *Proc. VLDB Endowment*, 1(1):28–41, 2008.

[163] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. USENIX OSDI*, 2010.

[164] sourceforge. Flamingo hadoop manager. `http://sourceforge.net/projects/hadoop-manager/`.

[165] Linkedin. Azkaban. `https://engineering.linkedin.com/data`.

[166] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1081–1090, New York, NY, USA, 2011. ACM.

[167] ornl. Titan. `http://www.olcf.ornl.gov/titan/`.

[168] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*, pages 137–150, 2004.

[169] Jeffrey Dean. Experiences with mapreduce, an abstraction for large-scale computation. In *PACT*, page 1, 2006.

[170] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09*, page 165, New York, New York, USA, June 2009. ACM Press.

[171] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[172] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. EuroSys*, pages 265–278, 2010.

[173] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy H. Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, pages 390–399, 2011.

[174] Apache Nutch. Nutch, 2010. `http://nutch.apache.org`.

[175] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in action*. Manning, 2011.

[176] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endowment*, 2(2):1626–1629, 2009.

[177] Daniel. Pigmix. https://cwiki.apache.org/confluence/display/PIG/PigMix, 2013.

[178] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[179] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proc. USENIX NSDI*, 2012.

[180] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Proc. IEEE ICDE*, pages 522–533, 2012.

[181] Price of intel 320 series 300 gb sata 2.5-inch solid-state drive on amazon, 2011. http://www.amazon.com/Intel-2-5-Inch-Solid-State-Drive-Brown/dp/B004UERMBC.

[182] Price of ocz revodrive 3 x 2 240 gb pci express 4 gb-s slim solid state drive on amazon, 2011. http://www.amazon.com/OCZ-RevoDrive-Express-Solid-State/dp/B005F30JBM.

[183] Dionysios Logothetis and Kenneth Yocum. Data indexing for stateful, large-scale data processing. In *Proceedings of NetDB*, 2009.

[184] Andy Konwinski and et.al . X-tracing Hadoop, 2008. Hadoop Summit.

[185] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proc. IEEE/ACM GRID*, 2004.

[186] Hyunsik Choi, Jihoon Son, YongHyun Cho, Min Kyoung Sung, and Yon Dohn Chung. Spider: a system for scalable, parallel/distributed evaluation of large-scale rdf data. In *Proc. ACM conference on Information and knowledge management*, 2009.

[187] Philip Schwan. Lustre: Building a file system for 1000-node clusters. In *Proc. Linux Symposium*, 2003.

[188] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. USENIX FAST*, 2002.

[189] Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, and Matthieu Dorier. Blobseer: Bringing high throughput under heavy concurrency to hadoop map-reduce applications. In *Proc. IEEE IPDPS*, 2010.

[190] Hadoop and Disparate Data Stores, 2012.

[191] Scale-out Storage Solutions for Hadoop, 2012.

[192] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX ATC*, pages 57–70, 2008.

[193] Arun C Murthy. Mumak: Map-Reduce Simulator. ASF JIRA MAPREDUCE-728, 2009.

[194] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Play it again, simmr! In *CLUSTER*, pages 253–261. IEEE, 2011.

[195] Fei Teng, Lei Yu, and Frederic Magoules. Simmapreduce: A simulator for modeling mapreduce framework. In *Multimedia and Ubiquitous Engineering (MUE), 2011 5th FTRA International Conference on*, pages 277 –282, june 2011.

[196] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: A mapreduce simulator in enabling cloud computing. *Future Generation Computer Systems*, (0):–, 2011.

[197] S. Hammoud, Maozhen Li, Yang Liu, N.K. Alham, and Zelong Liu. Mrsim: A discrete event based mapreduce simulator. In *Proc. FSKD*, 2010.

[198] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

[199] Guanying Wang, Ali R. Butt, Henry Monti, and Karan Gupta. Towards Synthesizing Realistic Workload Traces for Studying the Hadoop Ecosystem. In *MASCOTS*, 2011.

[200] Joe Elizondo and Samuel Palmer. Edge-based cloud computing as a feasible network paradigm. *Department of Computer Science University of Texas at Austin*, 2009.

[201] Shan Li, Joshua Sorchik, and Juzi Zhao. Balancing performance and power consumption in data-intensive computing clusters using job-type based scheduling. *Sort*.

[202] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proc. USENIX NSDI*, pages 21–21, 2010.

[203] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc. IEEE MSST*, 2010.

[204] R. Micheloni, L. Crippa, and M. Picca. Hybrid storage, 2013.

[205] Noshir Dhondy and David Petersen. Geographically dispersed parallel sysplex: The ultimate e-business availability solution. *IBM Corp*, pages 1–22, 2002.

[206] K. R. Krish et al. On the use of shared storage in shared-nothing environments. In *Proc. IEEE Big Data*, 2013.

[207] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[208] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proc. USENIX FAST*, 2010.

[209] Sam SD Chu and Calvin V Ho. Self-recovering erase scheme to enhance flash memory endurance, March 21 1995. US Patent 5,400,286.

[210] Linux man page. sar(1), 2013. `http://linux.die.net/man/1/sar`.

[211] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. IEEE MASCOTS*, 2011.

[212] Shengsheng Huang et al. Hibench: A representative and comprehensive hadoop benchmark suite.

[213] M. Poess and R. O. Nambiar. Energy cost, the key challenge of today's data centers: A power consumption analysis of tpc-c results. In *PVLDB*, 2008.

[214] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: helping disk arrays sleep through the winter. In *Proc. ACM SOSP*, 2005.

[215] Muthian Sivanthanu, Vijayan Prabhakaran, Andrea C. Arpaci-Duddeau, and Remzi Arpaci-Duddeau. Improving storage system availability with d-graid. In *USENIX FAST*, 2004.

[216] Alex Loddengaard . Cloudera's Support Team Shares Some Basic Hardware Recommendations, 2010. http://blog.cloudera.com/blog/2010/03/clouderas-support-team-shares-some-basic-hardware-recommendations/, accessed on Nov 13, 2012.

[217] Zhichao Li, Kevin M. Greenan, Andrew W. Leung, and Erez Zado. Power consumption in enterprise-scale backup storage systems. In *USENIX Fast*, 2012.

[218] Hadoop 0.23.0. ViewFs , 2011. http://hadoop.apache.org/docs/r0.23.0/api/org/apache-/hadoop/fs/viewfs/ViewFs.html, accessed on Nov 13, 2012.

[219] Yongqiang He and et. al. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE)*, 2011.

[220] Rodrigo Schmidt. Intelligent block placement policy to decrease probability of block loss, 2012. https://issues.apache.org/jira/browse/HDFS-1094, accessed on Nov 6, 2012.

[221] Yang Liu, Xiaohong Jiang, Huajun Chen, Jun Ma, and Xiangyu Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. *Advanced Parallel Processing Technologies*, pages 341–355, 2009.

[222] Dachuan Huang, Xuanhua Shi, Shadi Ibrahim, Lu Lu, Hongzhang Liu, Song Wu, and Hai Jin. MR-scope. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, page 849, New York, New York, USA, June 2010. ACM Press.

[223] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebooks warm blob storage system.

[224] KR Krish, Aleksandr Khasymski, Ali R Butt, Sameer Tiwari, and Milind Bhandarkar. Aptstore: Dynamic storage management for hadoop. In *Proc. IEEE CloudCom*, 2013.

[225] K. R. Krish, M.Safdar Iqbal, and Ali Butt. Venu: Orchestrating ssds in hadoop storage. In *Proc. IEEE Big Data*, 2014.

[226] Leo P Berenguel and John Reykjalin. Nonvolatile ramdisk memory, August 31 1993. US Patent 5,241,508.

[227] Michail D Flouris and Evangelos P Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4):281–293, 1999.

[228] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *Proc. ACM EuroSys*, 2009.

[229] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. Automated lookahead data migration in ssd-enabled multi-tiered storage systems. In *Proc. IEEE MSST*, 2010.

[230] An-IA ndy Wang. *The Conquest File System: A Disk/Persistent-RAM Hybrid Design for Better Performance and Simpler Data Paths*. PhD thesis, University of California Los Angeles, 2003.

[231] Chung H Lam. Storage class memory. In *IEEE ICSICT*, 2010.

[232] K.R. Krish, Ali Anwar, and Ali R. Butt. hats: A heterogeneity-aware tiered storage for hadoop. 2014.

[233] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proc USENIX OSDI*, 2000.

[234] Chris Gniady, Ali Raza Butt, and Y Charlie Hu. Program-counter-based pattern classification in buffer caching. In *Proc USENIX OSDI*, 2004.

[235] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinquan Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, 2010.

[236] Eaman Jahani, Michael J Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4, 2011.

[237] Michael J Cafarella and Christopher Ré. Manimal: relational optimization for data-intensive programs. In *International Workshop on the Web and Databases*, 2010.

[238] Ming-Yee Iu and Willy Zwaenepoel. Hadooptosql: a mapreduce query optimizer. In *Proc. EuroSys*, 2010.

[239] Sriram Rao, Benjamin Reed, and Adam Silberstein. Hotrod: Managing grid storage with on-demand replication. In *Proc. IEEE ICDEW*, 2013.

[240] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.

[241] S Radia and S Srinivas. Scaling hdfs cluster using namenode federation. *HDFS-1052, August*, 2010.

[242] Chuck Esterbrook. Using mix-ins with python. *Linux Journal*, 2001(84es):7, 2001.

[243] EC Amazon. Amazon elastic compute cloud (amazon ec2). 2010.

[244] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R Ganger, Michael A Kozuch, and Karsten Schwan. Robust and flexible power-proportional storage. In *ACM SOCC*, 2010.

[245] Electronic Educational Devices. Watts up pro, 2009.