A Resource Management Framework for Cloud Computing

Min Li

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
Kirk W. Cameron
Shravan Gaonkar
Xiaosong Ma
Cal J. Ribbens

March 25, 2014
Blacksburg, Virginia, USA

Keywords: Cloud Computing, MapReduce, Virtual Desktop Environment, Resource Management, I/O Reduction

A Resource Management Framework for Cloud Computing

Min Li

ABSTRACT

The cloud computing paradigm is realized through large scale distributed resource management and computation platforms such as MapReduce, Hadoop, Dryad, and Pregel. These platforms enable quick and efficient development of a large range of applications that can be sustained at scale in a fault-tolerant fashion. Two key technologies, namely resource virtualization and feature-rich enterprise storage, are further driving the wide-spread adoption of virtualized cloud environments. Many challenges arise when designing resource management techniques for both native and virtualized data centers. First, parameter tuning of MapReduce jobs for efficient resource utilization is a daunting and time consuming task. Second, while the MapReduce model is designed for and leverages information from native clusters to operate efficiently, the emergence of virtual cluster topology results in overlaying or hiding the actual network information. This leads to two resource selection and placement anomalies: (i) loss of data locality, and (ii) loss of job locality. Consequently, jobs may be placed physically far from their associated data or related jobs, which adversely affect the overall performance. Finally, the extant resource provisioning approach leads to significant wastage as enterprise cloud providers have to consider and provision for peak loads instead of average load (that is many times lower).

In this dissertation, we design and develop a resource management framework to address the above challenges. We first design an innovative resource scheduler, CAM, aimed at MapReduce applications running in virtualized cloud environments. CAM reconciles both data and VM resource allocation with a variety of competing constraints, such as storage utilization, changing CPU load and network link capacities based on a flow-network algorithm. Additionally, our platform exposes the typically hidden lower-level topology information to the MapReduce job scheduler, which enables it to make optimal task assignments. Second, we design an online performance tuning system, mrOnline, which monitors the MapReduce job execution, tunes the parameters based on collected statistics and provides fine-grained control over parameter configuration changes to the user. To this end, we employ a gray-

box based smart hill-climbing algorithm that leverages MapReduce runtime statistics and effectively converge to a desirable configuration within a single iteration. Finally, we target enterprise applications in virtualized environment where typically a network attached centralized storage system is deployed. We design a new protocol to share primary data deduplication information available at the storage server with the client. This enables better client-side cache utilization and reduces server-client network traffic, which leads to overall high performance. Based on the protocol, a workload aware VM management strategy is further introduced to decrease the load to the storage server and enhance the I/O efficiency for clients.

Dedication

Dedicated to my parents Delian Li and Hanci She, and husband Pengjie Lai, for their endless love, encouragement and support ...

Acknowledgments

I would like to express my deepest gratitude to my advisor, Ali R Butt, for everlasting encouragement, perpetual guidance and the vigorous and cooperative lab he has built. I never forget the nights throughout which he stayed up with us to meet paper deadlines. He has been always there to help, providing insightful advices and feedback for papers, presentations, not to mention that he supports us to attend a series of conferences to meet people and stay in the cutting edge of the literature. It is at DSSL Lab that I find myself absorbing sufficient nutrients needed to become a mature and independent researcher.

I would like to thank my advisory committee members, Shravan Gaonkar, Kirk W.Cameron, Xiaosong Ma and Cal Ribbens for their helpful and invaluable feedback on my graduate research. Particularly, I would like to convey my appreciation to Dr. Shravan Gaonkar for recruiting me as a summer intern at NetApp ATG. It was this first internship that opened up more opportunities for me and laid down the foundation of my dissertation topic. I had a productive and fabulous summer at 2010. I found Dr. Shravan an outstanding researcher who has strong hands-on system building skills which are crucial in system research. He also has a remarkably pleasant and easy-going personality. I would like to express particular thanks to Dr. Kirk Cameron for his advice on preparing my academic job interviews.

I was supported by IBM Ph.D. Fellowship for two consecutive academic years (2012-2014). I would like to thank IBM for providing me the financial assistance. I have spent three summers at IBM research centers where I developed my dissertation topic around resource management of MapReduce in the cloud. I would like to express my acknowledgement to the talented IBM researchers Dinesh Subhraveti, Liangzhao Zeng, Pin Zhou, Shicong Meng, Jian Tan, Li Zhang, Prasenjit Sarkar, Nicholas Fuller for insightful guidance of project defining, productive advices and energetic conversation.

I would like to thank my colleagues at Distributed Systems and Storage Laboratory (DSSL), particularly, Guanying Wang, Aleksandr Khasymski, Sushil Mantri, Yue(Kevin) Cheng, Abdul Hafeez, Krish K.R., Muhammad Mustafa Rafique, Henry Monti, Puranjoy Bhattacharjee,

Hyogi Sim, for their constructive comments, feedback, thought-provoking discussion. We have lovely times hanging out together in the Lab and attending conferences through the whole of my graduate years. I would like to thank my friends at Virginia Tech, especially, Chun-yi Su, Huijun Xiong, Heshan Lin, Yao Wang, Bo Li, Ji Wang, Kui Xu, Xiaokui Su, Jing Zhang, Shuaiwen (Leon) Song, Yingbei Wang, for all the fun we had together during all the entertaining events and fantastic moments.

Finally, I would like to thank my beloved parents, Delian Li, Hanci She for the immeasurable, unconditional love, encouragement and never-ending support since I was born. The consolation and encouragement from my father has always motivated and invigorated me to endure and survive any difficulties encountered during the graduate life. I would like to thank my husband, Pengjie Lai for the continuous love, reassurance and support throughout these years. Without them to be a source of my strength, I would have given up in the middle of my graduate study satisfying with a Master degree, making earning a Ph.D. degree to be a mission impossible task.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DHT Distributed Hash Table

DSF Distributed Software Framework

HC Hierarchical Clustering

IaaS Infrastructure as a Service

LHS Latin Hypercube Sampling

PaaS Platform as a Service

PM Physical Machine

Saas Software as a Service

UDF User-defined Function

VDE Virtual Desktop Environment

VM Virtual Machine

Chapter 1

Introduction

Cloud computing is a radical transformation underway in data centers and enterprises for increasing resource and administrative utilization and reducing energy consumption, acquisition and maintenance costs [1, 2]. Cloud computing has emerged as a model, offering infrastructure, platform and software as a service, from which users access resources from anywhere anytime on demand. It has been widely supported by large companies such as Amazon [3], Google [4], Microsoft [5, 6], Salesfore [7], Rackspace [8]. Small businesses especially startups have swiftly embraced cloud computing because of the benefits it offered. By hosting applications on a cloud platform, customers save the capital expense of investing and maintaining expensive servers. Cloud enables customers to access resources instantly and to provision, de-provision without intervention of third parties. Cloud service providers offer the illusion of infinite computing resources available on demand, allowing customers to require any quantity of resources at any time, which helps small businesses to scale up the infrastructure as their businesses expand. Moreover, because of the capability of per usage metering and billing, customers can provision additional resources as the load increases, and de-provision resources as the load decreases. The "pay-as-you-go" model releases customers from the burden of worrying about the maintenance of infrastructure, instead allows them to focus on the core business [9].

Cloud commonly refers to infrastructure, platform and software that are presented as a service remotely. In particular, cloud provides several service models. First one is infrastructure as a service (IaaS). Providers usually offer virtual machine resources and the capability to elastically provision resources to users from their large virtual machine (VM) pools running within data centers by pay-as-you-go model. Cloud users or enterprise employees install operating system images and their entire software stacks on allocated VMs. Examples are Amazon EC2 and Rackspace cloud. Platform as a service (PaaS) [1] delivers a computing platform typically including operating system, programming language execution environ-

ment, databases, and web servers. Application developers can develop and run their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers. A use case is MapReduce in the cloud. Examples are Amazon Elastic MapReduce and Window Azure. In the software as a service (SaaS) model, cloud providers install and operate application software in the cloud and cloud users access the software from cloud clients. Cloud users do not manage the cloud infrastructure or platforms on which applications are running. This eliminates the need to install and run applications on cloud users' own computers simplifying maintenance and support. Examples are Google App Engine. A use case is virtualized desktop environments(VDE) which decouple desktop environments from physical devices that are used to access them.

In this dissertation, we choose two typical cloud services: MapReduce in the cloud and VDE. MapReduce is an established framework for processing large-scale data-intensive applications. MapReduce allows developers to write applications that can easily scale to thousands of machines without worrying about the task distribution and failure recovery details. The MapReduce model helps businesses to process massive quantities of data in a reasonable time and extract valuable insights hidden. In particular, for many applications, such as converting archived media into a streaming format for Internet delivery, the processing is needed only once, and hence the resources required for processing are also needed only for a specific duration. Combining the MapReduce framework with the cloud provides a number of unique advantages. It is particularly appealing for organizations that need to analyze large amounts of data without having to acquire and manage large cluster resources. Users do not need to own cluster resources required to run jobs, which removes the entry barrier, enabling even small businesses to perform detailed analysis on their data. An organization can focus on its core business instead of being occupied by low level cluster maintenance. The cloud provides the flexibility of dedicating as many or as few VMs and storage resources as needed based on the required turnaround time. Users only pay for resources for the duration of time they are used.

VDE gains popularity in recent years because it delivers constant secure virtual desktop accesses to end users, simplifies desktop management and provides flexible dynamic IT infrastructure by decoupling applications, data, desktops with resource elasticity to meet businesses and end users' need. The VDE solution aims to provide the best option for both administrators and end users; it allows administrators to manage virtual desktops in a centralized manner, and let end-users access fully functional desktop environments through thin-clients by adopting a server-centric computing model. The creation, deletion and mi-

gration of virtual desktops and the deployment of new software update can be done with a few clicks while end users can access the virtual desktop anywhere any time securely. In VDEs, all the user data are stored in data centers and backed up through enterprise level redundant maintenance systems. Even if a user's device is lost or stolen, any critical data is less likely to be compromised. Moreover, it enables enterprises to dramatically lower the hardware, operational and the maintenance costs by centrally consolidating and dynamically managing virtual desktops.

The research in this dissertation aims to develop efficient resource management techniques for the two focused cloud services. The techniques proposed should be able to improve the application performance and scalability of the system by carefully managing the storage and compute resources and mitigating the I/O bottleneck. The goal of this dissertation is to assist in answering several open questions. How to design efficient resource allocation and management techniques for better resource utilization, better application performance? How to design optimization techniques to help improve the scalability of storage system?

In this chapter, we provide the necessary background for understanding the research done in this dissertation. More detailedly, Section 1.1 presents challenges and problems related to the two cloud services that we endeavor to address. Section 1.2 summarizes the research contributions of this dissertation. Section 1.3 describes an outline of the remains of this dissertation.

1.1 Challenges of Resource Management in Cloud Computing

Cloud resource management [10–12] serving as the core enabling technology for cloud computing orchestrates resources from a large number of computers and presents an uniform view to users and applications. It supports functionalities such as supporting a remote and secure interface for creating, destroying, configuring and monitoring virtual resources, dynamically managing resources, providing configurable resource allocation policies, elastically provisioning resources based on organizations' needs [13]. However, building a scalable cloud resource management system while meeting the requirements of flexibility, performance isolation, efficiency is challenging. Cloud management systems such as Amazon EC2 [14] have low machine utilization efficiency due to the low consolidation ratio by statically mapping one VM to a fixed number of physical CPUs and memories. Moreover, cloud providers allow

users to control resource requests through parameter configuration. It is important to assist users in easily issuing accurate resource requests without extra efforts. Finally, as the scale of cluster increases, the current resource management techniques can hardly catch up with the scale speed. The centralized storage infrastructure usually limits the number of VMs deployed in the cloud. Thus, it is critical for cloud service providers to design cloud management systems and tools that offer scalability, high performance, reliability, availability and security for rapid and wide adoption of cloud services.

The resource management of MapReduce in the cloud requires the collaboration between users and MapReduce service providers. From the providers' perspective, they should be able to place the data and VM effectively to avoid unnecessary network traffic and guarantee efficient resource utilization; from the users' perspective, they need to provide accurate resource requests which typically depend on application characteristics. Thus, we approach the problem from both cloud providers' perspective 1.1.1 and from users' perspective 1.1.2. For VDEs, one of the key problem is that enterprises suffer from expensive centralized storage system provisioning for peak loads 1.1.3. In the following, we present an overview of problems and challenges involved in the two targeted cloud services that we address in this dissertation.

1.1.1 Performance Degradation of MapReduce in Virtualized Clouds

While cloud offers great promise, MapReduce suffers performance degradation because of several reasons discussed as follows. The storage infrastructure of existing cloud environments is poorly suited for MapReduce computation. Clouds are typically built on commodity clusters with node-local disks for their cost-effectiveness and scalability. Several issues affect the turnaround time of MapReduce jobs running in these cloud environments.

First, *running MapReduce jobs in the cloud has an expensive ingestion phase*, where the dataset needs to be copied from a central persistent store into the compute cluster for processing. For large datasets, ingestion represents a significant portion of the turnaround time. Moreover, clouds feature a stateless model where any data and VMs copied to the physical/hypervisor cluster are discarded once the job is completed. Subsequent jobs require transferring the data again.

Alternatively, it may be possible to have the MapReduce tasks access the data directly from

the remote store via suitable remote data access protocols such as NFS, iSCSI [15], or FibreChannel [16]. Such remote access has several disadvantages. For one, all data would have to be accessed over the network. MapReduce model achieves its efficiency by ensuring that tasks can access their data locally. Thus, fetching data over the network severely affects job performance. Furthermore, the bisectional bandwidth between the compute cluster and the central store can easily become a bottleneck. A large number of tasks, all accessing their data from the central store, can quickly saturate the link and render the system inefficient.

A further alternative is to co-locate the data with the compute cluster. However, spreading out the data across the local disks of cluster nodes constrains the scheduling choices available for placing VMs. VMs accessing the data located on a particular node must all be placed on that node, but other constraints such as the amount of memory or licenses may not permit this. Providing reliability of persistent data located on the hypervisor cluster is also a challenge. Data stored on a centralized storage device is typically protected from disk failures through internal replication. Providing a similar replication facility across disjoint local file systems storing the data is difficult. A cluster file system may be used to combine the local storage attached to individual nodes in the cluster, but most existing cluster file systems are designed for a central storage model in a storage area network (SAN), and perform poorly on local storage [17]. For example, cluster file systems typically stripe files across all available disks in the cluster to maximize throughput, whereas such a strategy limits the performance in commodity clusters where network is the bottleneck.

Similar issues also apply to the VM images that compose a MapReduce job. The virtual image files need to be copied to the hypervisor nodes before starting a job, which introduces an exceptionally high startup latency. As before, running directly from the remote storage is not a scalable solution while co-locating the images with the cluster limits scheduling choices.

Second, *the cloud masks the physical topology of the underlying infrastructure*, which can potentially inhibit optimal scheduling of MapReduce tasks. The MapReduce model is designed for and leverages information from the native clusters to operate efficiently, whereas the cloud presents a virtual cluster topology. For instance, the VMs associated with a job may be placed across multiple racks. However, this information is not typically visible to the application. Furthermore, the cloud may also change the initial assignment by migrating the VMs to different nodes in the cluster based on runtime load and other constraints. While these functions add flexibility, they also make application-level scheduling challenging. This results in two *placement anomalies*: (1) Loss of data locality, where a task may be placed away from the physical location of its data; and (2) Loss of job locality, where a task may be

placed away from the physical locations of other tasks with which it communicates. Map-intensive jobs are adversely affected by loss of data locality, and reduce-intensive jobs are impacted by loss of job locality.

Third, *the multi-tenant cloud environment may result in interference between MapReduce applications* and other applications sharing the environment. Scheduling decisions made at the beginning of the job may become invalid during the course of the job, when VMs are migrated around or due to changing workloads. The optimal allocation of resources might become suboptimal leading to poor performance.

1.1.2 Inaccurate Resource Requests through Job Parameter Configuration

MapReduce framework exposes resource requesting knobs through parameter configuration to users allowing them to request the amount of resources needed within a given quota. While MapReduce framework enables users to scale up applications easily, writing good MapReduce applications requires specialized system-level skills and extra effort as users also have to provide application specific job/system parameters requesting needed resources. These parameters are crucial and affect performance significantly. Inaccurate resource requests can lead to resource underutilization or over-utilization and performance degradation. For example, consider the configuration parameter, "io.sort.mb" that controls the amount of buffer memory to use when sorting files, and thus setting it to suboptimal values can lead to unnecessary I/Os and consequently increased task running time. Moreover, different applications require different values for "io.sort.mb" depending on the HDFS [18, 19] block size and the map task output size. Similarly, applications vary in demands, e.g., MapReduce application Grep [20] requires less sort space than Terasort [21] since Grep usually outputs fewer data than Terasort in map phase. The importance of such performance tuning is highlighted by a simple web search turning up a list of best practices for MapReduce tuning guides [22–26]. These documents show multiple orders of performance gain for applications under tuned parameters compared to the default settings. Recent research, e.g., Starfish [27], shows that MapReduce application performance depends on the size and content of data sets, job characteristics, the cluster hardware configuration, and more importantly configuration parameters.

Unfortunately, MapReduce job parameter tuning is a daunting and time consuming task, mainly due to the fact that the parameter configuration space is huge. The number of per-

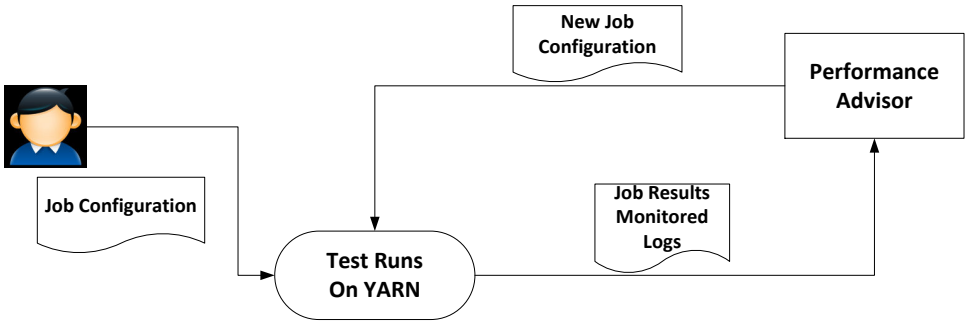

Figure 1.1 The traditional offline performance tuning for MapReduce applications.

formance related parameters in Hadoop [19] is more than 70. Moreover, it is difficult for a user to figure out the optimal value for a parameter without first having a deep understanding of the MapReduce application characteristics. To address this challenge, the current approach is to use offline performance tuning technique. As shown in Figure 1.1, traditional offline tuning first comes up with a configuration based on a default setting or a rough understanding of application characteristics. Next, test runs of the application are executed with profiling enabled and data such as job performance counters, system monitoring logs and the profiling outputs is collected. The user then feeds those results to a performance advisor or manually analyzes the statistics and generates a new configuration. The process is usually repeated for multiple runs until a desired configuration is reached. The selected configuration is then employed while running the application on production clusters.

There are multiple drawbacks of the above traditional performance tuning for MapReduce. First, the process is time consuming since it requires many test runs, and each run can only try a single configuration. This is further exacerbated when the application involves long running jobs. Second, the offline approach is not cost effective if the tuning is done for an application that will only run for few times or perhaps just once. The users would rather simply run their applications without tuning, leading to inefficient resource utilization. In addition, as shown in Starfish [27], the optimal configuration also depends on data sets and cluster hardware configuration. So, the users would have to adjust the parameters each time they change input data sets or run applications on different clusters. Moreover, no one configuration fits for all tasks within one job. MapReduce jobs also commonly exhibit data skew [28] that requires different amount of resources based on the different sizes of data being processed. Finally, the traditional offline tuning statically tunes the parameters once and use the configuration throughout the whole life cycle of a MapReduce job. However, the job characteristics and cluster utilization are dynamic, and static tuning cannot adapt to such variations and thus cannot avoid performance-degrading cluster hot spots.

1.1.3 Storage Limitation of VDE

Within the VDE, the storage infrastructure is typically realized using shared storage box(es) that offer management features preferred by system administrators [29]. These rich features include high availability, fault tolerance, distributed resource scheduling, site recovery management, storage migration, etc., which are not easy or even impossible to achieve using just local storage.

However, the virtual desktop deployment suffers significant capital costs from enterprise storage. In order to support the high scalability enabled by virtualization technology, the shared storage must be provisioned in a way that is easy to scale. Moreover, the shared storage needs to support not only average IOps from end users, but also the peak load such as boot storms, login storms, virus scan storms, and testing developing storms. For example, boot and login storms usually happen at 9 am on weekdays and virus scan storms happen at 3 am [30]. While the average IOps from light users are usually 8-10 IOps and from heavy end users is 14-20 IOps, the login process generates 90-100 IOps per end user on average, which is around 5 times higher than the load from a heavy end user [31]. Similarly, the login storm pushes the peak load much higher, which has to be provisioned in order to provide smooth user experience.

Researchers and vendors have observed that there is a lot of duplicate data within VDEs. For instance, the virtual images usually are created using the same golden images and the virtual desktops typically install the same set of applications such as anti-virus software and web browsers. Based on this observation, the I/O reduction techniques including dedupbox [32]; atlantis ILIO [33]; Capo [29] have been proposed to reduce the duplicated I/O load from the shared storage system, and hence improve the storage efficiency. However, those current approaches require adding on-wire deduplication boxes or cannot detect duplicate data on write path.

The effectiveness of all the above-mentioned techniques depends on the amount of duplicated data accessed by VMs running on the same physical hosts. While VMs are usually placed and managed by a centralized VM manager, suboptimal VM placement can lead to reduced (or preclude) common data accesses by VMs on the same physical host and thus results in less I/O reduction. For example, a naive VM placement algorithm that places virtual desktops belonging to employees from a payroll department and virtual desktops belonging to employees from a software development department indistinguishably reduces the opportunity to detect common accesses and affects I/O reduction efficiency. In contrast, placing

virtual desktops of the payroll department on one set of physical hosts separately from those of the software development department offers better reduction in I/O.

1.2 Research Contributions

This dissertation proposes an innovative resource management framework for two focused cloud services, namely MapReduce in the cloud and virtual desktop environment. While designing the planned framework, we propose innovative approaches and systems to tackle the challenges of accurate resource requesting through parameter tuning, VM/ data management for MapReduce in virtualized clouds and storage scalability of VDE. In the following, we highlight research contributions that we make in this dissertation.

1. We design, develop and evaluate a cloud resource manager, **CAM**, to maximize the locality for MapReduce in the cloud. (1) Data placement: Data is placed within the cluster based on offline profiling of the jobs that most commonly run on the data. Rather than accommodating an arbitrary data placement, strategically placing the data can significantly improve locality. (2) VM/job placement: For a given job, CAM selects the best possible physical nodes to place the set of VMs that represent the job. (3) Task placement: In order to further minimize the possibility of a placement anomaly, CAM exposes, otherwise hidden, *compute, storage, and network* topologies to the MapReduce job scheduler such that it makes optimal task assignments. This is crucial as, for example, what appears to be a directly attached local disk within a VM could in fact be physically located on a different node. CAM reconciles resource allocation with a variety of other competing constraints such as storage utilization, changing CPU load and network link capacities using a flow-network-based algorithm that is able to simultaneously satisfy the specified constraints. Each placement decision not only considers the existing data and VM assignments in the cluster, but also evaluates the cost of readjusting existing assignments in response to data movement and VM migration to derive the best net configuration possible.

2. We investigate how to easy users' burden to issue accurate resource requests through dynamic parameter tuning. Specifically, We design and implement a task-level dynamic configuration framework,**mrOnline**, based on YARN [34], the second generation of open-source MapReduce implementation. mrOnline enables different configuration

for each map and reduce task, thus providing for high performance. We then design a gray-box based smart hill-climbing algorithm to systematically search through the MapReduce parameter space. We support aggressive and conservation tuning strategies for two use cases. To improve the search quality and reduce convergence iteration, we propose tuning rules for key parameters. We evaluate mrOnline on YARN and present an experimental performance evaluation on a 19-node cluster. Our results demonstrate that compared to default YARN settings, our approach achieves an efficiency improvement of 22% for dynamically tuned applications. Moreover, for applications that run multiple times, mrOnline can expedite the test runs and reduce job execution time by up to 30%. Our results show that mrOnline offers an effective means to improve MapReduce application performance.

3. We propose a holistic cache system to improve the scalability of storage system infrastructure for VDEs. In particular, we examine (1) host side caching, (2) data transfer within hosts and between storage server and hosts, and (3) storage server side deduplication in a holistic manner, and realize a software-only solution that obviates the need to provision for peak loads without employing extra memory or adding on-wire de-duplication boxes. We design and implement **SeaCache** in the context of the NFS protocol for client/storage interactions in consolidated data centers offering a novel holistic approach to support consolidated workloads. More specifically, we present read/write content sharing algorithms and a collective cache in the context of an integrated framework and compare the alternate algorithms using real-world traces.

4. To improve the I/O workload similarities between VMs thus guaranteeing the effectiveness of deduplication techniques like SeaCache for VDE, we design, develop and evaluate a VM placement and migration system, **SMIO**, in Xen platform. SMIO detects I/O similarity between VMs, migrates the VMs based on the I/O similarity benefit and the migration costs. We also propose a multi-phase filtering technique to improve the quality of our clustering result and the final migration plan. Evaluation using trace driven simulation shows that SMIO can effectively detect the similarity between different sets of VMs and improve the performance of storage system.

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, from service providers' perspective, we introduce a topology-aware min-cost-flow based resource manager, CAM, which aims to alleviate placement anomalies and performance degradation of MapReduce jobs when hosting MapReduce clusters in the cloud. We discuss how CAM adopts a three level approach to mitigate the inefficient resource allocation and placement anomalies. We also present an evaluation using both micro-benchmark and macro-benchmark to show the effectiveness of CAM compared against a state-of-the-art resource manager. In Chapter 4, we explore how we can help users to issue accurate resource requests by enabling dynamic online performance tuning from users' perspective. We describe how we enable dynamic task level configuration and systematically search through the configuration space using a gray-box based hill climbing algorithm. To increase the convergence speed of our search algorithm, we also incorporate a set of MapReduce specific tuning rules. An implementation and evaluation using a 20-node cluster demonstrate the effectiveness of our approach. In Chapter 5, we explain the design of SeaCache to improve the storage scalability and efficiency of centralized storage system in VDEs. We explore different caching protocols and compare their efficiency in terms of I/O reduction, latency reduction and system overhead. We present a detailed implementation and simulation methodology to show that SeaCache effectively reduces I/O bandwidth consumption. In Chapter 6, we study how workload similarity aware VM placement can help improve the I/O reduction efficiency of techniques such as SeaCache. We describe a hierarchical clustering algorithm to cluster VMs with high I/O similarities together and a greedy migration algorithm with the goal of minimizing migration overhead. Finally, we details a series of experiments to illustrate that SMIO can help enhance I/O reduction efficiency. We then conclude the dissertation in Chapter 7 including a discussion of future directions based on our resource management framework.

Chapter 2

Related Work

As this dissertation focuses on different aspects of resource management in clouds, namely the VM and data management, MapReduce performance tuning for efficient resource utilization and storage scalability of VDEs, we summarize the state of the art that is close related to these problem spaces in this Chapter. More specifically, we review the resource management and VM management approaches, MapReduce performance tuning techniques followed by a discussion of I/O de-duplication mechanisms adopted for improving scalability of storage system.

2.1 Resource Management for MapReduce in the Cloud

We present a number of research efforts related to resource management techniques for hosting MapReduce in both native clusters and virtualized clusters in this section, which is pertinent to our topology aware resource manager CAM described in Chapter 3.

Resource allocation for MapReduce in the cloud has received a lot of attention in recent works [35–39]. The project closest to our proposed resource manager, CAM, is a resource allocation system called Purlieus, developed by Palanisamy et al. [35]. Purlieus arrives at a job-local data and VM placement solution according to heuristics specifically developed for different job types, such as Map-input or Reduce-input heavy. The system defines both data and VM locality, as well as physical machine load, which are similar to the notion of *VM closeness* and *Hotspots* in CAM. Unlike Purlieus, CAM employs a min-cost flow based approach, which can consider both VM migration as well as delayed scheduling to arrive at a global optimal placement. Additionally, CAM optimizes for *Storage utilization*, which allows it to do data, as well as CPU utilization, load balancing, consequently improving the overall VM placement. Moreover, CAM uses both the actual location of data and network topology

as its inputs, whereas Purlieus relies on the virtual topology which may be different from the physical topology.

LATE [36] improves MapReduce performance in the cloud by performing effective speculative execution to reduce the job running time, while ignoring speculative task locality. CAM is different from LATE in that it couples the data placement, VM placement, and task placement to systematically improve data locality for MapReduce in the cloud.

There are several efforts that focus on MapReduce task scheduling in terms of data locality and fairness. Mantri [40] manages outliers in a resource and cause aware manner on native cluster. Delay scheduling [41] target fairness scheduling while maximize the map tasks locality on native clusters by delaying a task multiple times. Although delay scheduling offers a simple technique to provide better locality, it does not consider global scheduling, thus it loses the opportunity for achieving better performance. Moreover, the effectiveness of delay scheduling relies on the assumption that most tasks comprise of either small or long jobs. Quincy [39] uses similar graph techniques, but it differs from CAM in terms of problem space and associated flow network construction. Quincy strikes a balance between fairness and data locality, while CAM focuses on optimizing data/VM placement of MapReduce applications in the cloud. As a result, the factors encoded in the flow graph (VM closeness, Hotspot, etc.) are fundamentally different from that of Quincy's.

2.2 Virtual Machine Management

A plethora of VM placement and migration techniques are proposed in the cloud to optimize for minimizing network traffic, energy, meeting SLA requirement and so on [42–47]. The VM placement problem is essentially a bin-packing problem for which various heuristics are applied.

In order to support efficient consolidation for reducing power consumption within virtualized data centers, Verma [48] et al. propose two static consolidation approaches, correlation based placement and peak clustering based placement. Correlation based placement tries to provision individual applications based on off-peak demand while carefully controlling the SLA violation risk, and to add co-location constraints of positively correlated applications to prevent placement on the same physical hosts. However, this approach yields unbalanced workloads across active servers and is not scalable. Peak clustering based placement uses a

two-level envelop of original time series to cluster workload and place applications sorted in power-efficient order into minimal number of physical hosts.

Entropy [49] targets the problem of placing VMs in minimum number of physical hosts, while mitigating migration overhead to save energy. Entropy first formulates the problem as a n dimensional bin packing problem (constraint satisfaction problem), which is solved using Choco library, and then calculates a migration plan. This work also attempts to reduce the number of migrations by considering a migration cost model.

Recon [50] aims to minimize the number of physical hosts in a virtualized data center by analyzing historical data. It formulates the placement as an optimization problem and utilize AMPL and CPLEX to solve the problem. The solution attempts to discover the CPU utilization that exhibits complementary behavior. The goal here is to provide a tool for users to explore the solution space offline instead of a tool to help placing VMs for cloud providers as is the focus of SMIO discussed in Chapter 6. Recon does not discuss the scalability of the approach and focuses only on CPU utilization.

With the goal of minimizing the overall network traffic among VMs within data centers, Xiaoqiao et al. [51] formulate the network traffic aware VM placement as an optimization problem, and prove it as an NP hard problem that is computationally equivalent to quadratic assignment problem. In order to solve the problem, they designed a two-level hierarchical algorithm that assumes that the traffic pattern among VMs are known, and uses it to map the VMs to a cluster followed by mapping the VMs to each physical host. It is an approximation heuristic based on network traffic pattern and topologies in data centers, which assigns VM pairs with heavy communication traffic to physical hosts with fast connection. With the same goal, Starling [52] proposes to monitor network traffic by maintaining a dynamic affinity matrix of exponential average over its past values and migrate VMs within a time window using distributed greedy heuristics. When the benefits of rearrangement exceed a threshold (estimated based on memory size and available bandwidth link), the migration is executed.

Due to the dynamic workload fluctuation within virtualized data centers, it is important to detect and remove the hot spots by migrating VMs. Sandiper [53] discusses a system that detects hot spots and migrates VMs based on a greedy heuristic algorithm to alleviate hot spots. Two approaches to detect hot spots include black-box, which is OS and application unaware, and gray-box, which on the other hand gathers information from OS and applications. Black-box approach is unobtrusive, however, gray-box approach is more effective. The main idea of the greedy migration algorithm is to move the VMs from the most loaded

host to the least loaded host while at the same time trying to minimize data copied in the migration process.

The above techniques are complementary to our proposed VM placement method, SMIO, in that none of the works aims to optimize I/O performance by placing VM strategically. Given the different goal of SMIO, we formulate the problem differently. We mainly consider the I/O workload similarity when grouping and placing VMs, however, the mechanisms above can be incorporated into our framework. Moreover, such work is different from CAM, the system optimized for running MapReduce in the cloud because it does not consider job characteristics which are critical for managing MapReduce instances in the cloud.

2.3 Parameter Tuning of MapReduce Framework

This section reviews the research related to our dynamic online performance tuning system, mrOnline introduced in Chapter 4, which aims to assist users in requesting accurate resource needs through parameter configuration for running MapReduce applications. We group the related works into three categories: 1) research works focusing on MapReduce configuration parameter tuning which solve a similar problem space as mrOnline; 2) projects targeting MapReduce performance tuning from other aspects such as optimizing UDF data flows; 3) research papers relevant to performance tuning from other areas including network parameter configuration, etc.

MapReduce Configuration Parameter Tuning: There are several works [27, 54, 55]. that have focused on MapReduce job configuration tuning in recent years. Herodotos et al. [27, 54, 55] proposed a cost based optimization technique to help users identify good job configurations for MapReduce applications. Their system consists of a profiler to get concise statistics including data flow information and cost estimation, a what-if engine to reason about the impact of parameter configuration settings, and a cost based optimizer to find good configurations through invocations of the what-if engine. The effectiveness of their system depends on the accuracy of the what-if engine that uses a mix of simulation and model based estimation. mrOnline is different from this work in that mrOnline finds desirable configuration parameters through real test runs on real systems. Additionally, mrOnline uses task level dynamic configuration to avoid multiple what-if iterations, and unlike such prior approaches is also able to adjust to dynamic cluster runtime status, e.g., network congestion or I/O congestion.

Gunther [56] is another offline tuning method that uses a Genetic Algorithm to identify good parameter configurations, tries one configuration per test run, and can take $20 - 40$ test runs. In contrast, mrOnline can tune within a single job run. Moreover, mrOnline is a gray box approach that effectively exploits MapReduce runtime statistics, while Gunther is a black-box approach. In addition, we identify the two use cases; aggressive tuning aims to reduce the number of test runs, while conservative tuning can help improve the performance of jobs that only run once. Gunther cannot help in either case.

AROMA [57] aims to automate the resource allocation and job configuration for heterogeneous clouds and is aimed at satisfying SLAs while minimizing cost. AROMA uses a two-phase machine learning and optimization framework based on support vector machine based performance models. The two phase technique is composed of a offline and online phase. The offline phase classifies executed jobs using k-mediod clustering algorithm using CPU, network, and disk utilization patterns, while the online phase captures the resource utilization signature of tested applications. Finally, AROMA find near optimal resource allocation and configuration parameters based on a pattern matching optimization method. Compared with mrOnline, AROMA does not support dynamic configuration. Moreover, AROMA has to first collect application resource utilization signatures before finding a near optimal configuration. This is not suitable for jobs that run once.

Parameter tuning guides [22–26] are also proposed by industry and vendors to help MapReduce non-experts to set desirable values for their applications. However, these tuning guides are based on heuristics. The burden is still on the end users to try out multiple parameter combinations, which is time consuming and cumbersome as discussed in Section 1.1.2.

MapReduce Performance Tuning: Performance tuning of MapReduce framework [58–61] has gained a lot of attention from industry and research. MANIMAL [58] focuses on the efficiency of query processing of MapReduce framework and utilizes static program analysis techniques on user-defined functions (UDFs) to detect standard query optimization opportunities. To bridge the performance gap of MapReduce and parallel DBMS, Hadoop++ [61] tries to inject optimization into UDFs, which makes query processing pipeline explicit and present it as DB style physical query execution plan. This work has a different focus than mrOnline. SUDO [62] analyzes UDFs to identify beneficial functional properties to optimize data shuffling for MapReduce frameworks by utilizing program analysis techniques. PerfXplain [60] provides a tool for non-expert users to tune MapReduce performance. This tool auto-generates an explanation for the queries comparing two jobs, which can help identify the reasons why inefficiency or unexpected behavior happens. However, this work does not

provide clear guidelines of what job configuration parameters should be used. Jiang et al. [59] provides a performance study of MapReduce, pinpointing factors that impact MapReduce performance including I/O, indexing, record decoding, grouping schemes and block level scheduling in database context. Although these works share with mrOnline the goal to improve MapReduce application performance, these systems differs from mrOnline because of different optimization aspects and different targeted environments. Moreover, to the best of our knowledge, mrOnline is unique in its focus on YARN-based systems.

Simulation based performance tuning [63, 64] techniques have also explored. Our own previous work, MRPerf [63], utilizes a simulation methodology to capture various factors that impact Hadoop performance. Similarly, Mumak [64] is designed as a MapReduce simulator for researchers to prototype features and predict their behavior and performance. These projects do not tune configuration parameters as such and only provide means to estimate application performance on given configurations.

Parameter Tuning in Other Areas: A number of search techniques are proposed to find good configuration with high probability [65, 66] in other research areas as well. Recursive random search [65] is a black box optimization approach that employs a heuristic search algorithm for tuning network parameter configurations. Smart hill climbing, designed for server parameter tuning, is another black box optimization approach that is designed to improve the recursive random search algorithm. Smart hill climbing adopts a weighted LHS technique to improve the random sampling on the first phase. Moreover, the algorithm learns from past and searches the space using steepest descent direction and improves the search efficiency. The tuning algorithm of mrOnline is inspired by smart hill climbing algorithm. However, since the targeted problem is different, the challenges faced in mrOnline are different from the above algorithms.

iTuned [67] concentrates on tuning database configuration parameters by adaptive sampling and uses an executor to support online experiments through a cycle staling paradigm. This approach is not suitable for MapReduce systems. JustRunIt [68] is an experiment based management system for virtualized data centers. It shares with mrOnline the goal of tuning parameters using actual experiments that are cheaper, simpler and more accurate than performance models or simulations. However, the approach is not simply applicable to MapReduce.

2.4 I/O Deduplication

Optimization Focus	Strategy	Research Projects
Storage Server	De-duplication	Venti [69], REBL [70], IBM N Series SS [71], De-dup FS [72]
	Optimized Index Structures	De-dup FS [72], Sparse-Indexing [73], Foundation [74]
	Back Reference Tracking	Backlog [75]
Host / Storage Server Interactions	Caching Hints	Exclusive caching [76], Write-hints [77], X-RAY [78]
	Hash-value Passing	CASPER [79], DeDe [80], Pastiche [81]
	Optimize On-wire Transfers	LBFS [82], TAPER [83], Capo [84]
Host	Memory De-duplication, Compression	Difference Engine [85], ESX Server [86], Satori [87], I/O De-dup [88]
	Cooperative Caching	Cooperative Cache [89], LAC [90], Shark [91]

Table 2.1 Classification of related research.

In this section, we classify several prior research works (Table 2.1) related to our research, especially SeaCache 5 and SMIO 6, in optimizing the I/O path based on where the optimizations are made, namely: 1) at the host; 2) at the storage server; or 3) during the interactions between the host and the storage server.

Storage Systems De-duplication Management: A number of works have explored identifying and removing redundancy from stored data to optimize storage usage. Venti [69] utilizes SHA hashes on fixed-sized blocks of data to avoid having to store multiple copies of duplicate data for archival storage. REBL [70] introduces the idea of super-fingerprints to further optimize the amount of data needed for identifying duplicates, thus improving performance. The IBM N Series Storage Systems [71] offers a near-line version of de-duplication techniques in a real system implementation. Similarly, De-duplication File System [72] utilizes techniques such as compact in-memory data structures for identifying duplicates, and improved on-disk locality to yield high efficiency.

Research on the storage server side optimizations also include design of advanced data structures to improve de-duplication efficiency, e.g., in large-scale file systems [72], stream processing [73], and user storage [74], etc. Finally, Backlog [75] offers means for efficiently supporting features such as defragmentation and migration in the presence of de-duplication.

These techniques are complementary to SeaCache. We focus on communicating the data de-duplication information between the storage server and the hosts. SeaCache optimizes the data transfer and latency by leveraging this de-duplication information without any dependency on the underlying topology (primary or backup).

Host / Storage Server Interaction Optimizations: The prior art on obtaining better cache utilization at hosts and storage servers by treating them as an integrated unit offers optimizations, such as exclusive caching [76], sharing write-hints [77], and inferring accesses, e.g., with X-RAY [78], which are complementary to our work. The integrated caching frameworks focus on reducing duplicate data between different tiers of storage caches. However, in SeaCache, the main focus is on avoiding transfer of duplicate data between storage servers and client VMs on hosts.

Distributed hash management techniques such as those employed in CASPER [79] and Pastiche [81] deal with storing content hashes at the hosts so that they can choose the most cost effective replica from among a set of storage servers. Similarly, de-centralized de-duplication (DeDe) [80] uses techniques where a set of hosts communicate with each other to de-duplicate data. Such techniques can be leveraged in SeaCache both for transferring de-duplication information between the host and storage server, as well as for host buffer management.

Finally, the on-wire bandwidth reduction techniques, e.g., in LBFS [82] and TAPER [83], are different from SeaCache in that these techniques keep track of the data being sent between the hosts and the storage servers and try to not send duplicates. These techniques are not integrated with the host buffer management techniques, and so the hosts can continue to send data requests to the storage servers even if they have the data cached, and similarly the storage server will do the necessary processing associated with a data request even if it has previously sent the data to the host. Capo [84] leverages the fact that most of the VM disk images are the linked clones from a small set of "golden images" and uses a bit-map to eliminate the duplicate read requests. However, unlike SeaCache, Capo cannot detect duplicate reads outside of golden images or duplicate writes.

Host Cache Management: The host side client caching research primarily consists of how to compress and de-duplicate the client cache (e.g., Difference Engine [85], ESX Server [86], and Satori [87]), and how to avoid sending duplicate read data request to disks (e.g., I/O De-duplication [88]). The cache replacement algorithms and the transferring of cache state to the storage server techniques employed in SeaCache are complementary to the previous host side cache de-duplication/compression strategies and can be employed in that context. Moreover, in SeaCache, the de-duplication algorithms on host and storage server can share

de-duplication information to eliminate the CPU intensive recomputation of content sharing attributes.

These I/O reduction mechanisms serve as a base for our proposed workload similarity-aware VM placement, SMIO, in that simply adapting our technique can not immediately benefit the storage system performance without using the I/O reduction approaches. On the other hand, the effectiveness of the mechanisms all depends on the I/O workload similarity among VMs. In the above works, the efficiency would suffer if the I/O workload similarity in the same physical host is low. In contrast, SMIO aims to improve the I/O similarity among VMs running on the same physical host, thus improves the efficiency and scalability of the storage system.

Cooperative Caching: Cooperative caching techniques focus on designing efficient eviction algorithms and meta-data indices to aggregate distributed client caches as a unified cache and to facilitate fast lookup. N-chance forwarding [89] assigns more weight to singlets that have only one copy of data in the cache by forwarding singlets to random peers. LAC [90] forwards the evicted data block to peers based on data reuse distance and dynamic client synchronization. Shark [91] designs a locality aware distributed index to enable clients to locate nearby copies of data.

These techniques are orthogonal to SeaCache, as SeaCache introduces the de-duplication concept into the cooperative cache and focuses on how cooperative caching can help to reduce I/O bandwidth consumption. In our SMIO system, we offer an alternate solution for mitigating the I/O demand to storage server other than use cooperative cache. Moreover, in the case where a group of VMs have similar I/Os for a relatively long time period, our placement will eliminate the associated network traffic (at the cost of one-time migration), which cooperative cache can not.

2.5 Chapter Summary

In this chapter, we discuss the related work on resource management of MapReduce in the cloud, the virtual machine management, the parameter tuning and I/O reduction technique for VDEs. The framework presented in this dissertation targets to provide efficient resource management approaches for both MapReduce in the cloud and VDEs with the main objective of improving resource utilization efficiency and application performance in a transparent and efficient manner.

Chapter 3

Topology-aware Minimum-cost-flow Based Resource Management

Hosting MapReduce platform on virtualized clouds enables enterprises especially startups to focus on the core business without acquiring or maintaining any large cluster resources. However, as MapReduce is originally designed to run on bare metal clusters, it suffers from performance degradation due to placement anomalies and resource scheduling inefficiency. Reasons come from the fact that clouds overlay network and storage topology information from MapReduce framework and encounter scheduling conflicts and interference between MapReduce applications.

In this chapter, we present CAM, a platform that is designed to host MapReduce applications in virtualized clouds. CAM provides a cluster file system that supports an uniform file system name-space across the cluster by integrating the discrete local storage of the individual nodes. The shared file system enables a VM to be placed on any cluster node or subsequently migrated as necessary. We leverage GPFS [92] in CAM to query and specify the physical locations of an image and its replicas, which can then be used for CAM-directed placement of VMs and data. CAM avoids the placement anomalies with an innovative resource scheduler for the cloud, especially designed for improving the performance of MapReduce jobs. CAM reconciles both data and VM resource allocation with a variety of competing constraints, such as storage utilization, changing CPU load and network link capacities. CAM uses a flow-network-based algorithm that is able to optimize MapReduce performance under the specified constraints – not only by initial placement, but by readjusting through VM and data migration as well. Additionally, CAM exposes, otherwise hidden, lower-level topology information to the MapReduce job scheduler so that it makes near-optimal task assignments.

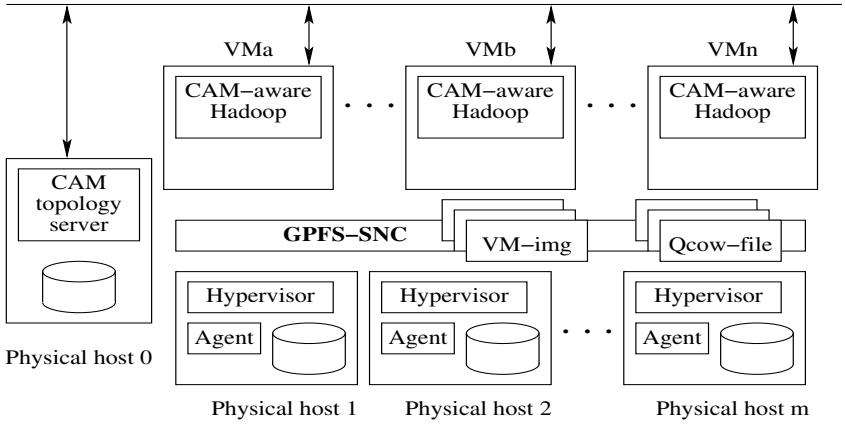

Figure 3.1 CAM architecture components.

3.1 Architecture

CAM is designed as an extension to IBM ISAAC product [93]. ISAAC implements key cloud functions such as creating and deleting VMs and their persistent volumes, placing the VMs based on load and capacity, maintaining availability of cloud services through clustering and fail-over mechanisms. The architectural components of CAM are implemented as extensions to related counterparts in ISAAC. In particular, we have integrated ISAAC with the GPFS-SNC [92] file system to provide a suitable cluster file system needed by CAM, and have extended ISAAC to support data and VM placement based on techniques we describe in Section 3.3. Figure 3.1 illustrates the components of CAM and their interactions when deployed in a cloud environment. The physical resources supporting the cloud consists of a cluster of hypervisor (physical) nodes with local storage directly attached to the individual nodes.

3.1.1 GPFS-SNC Storage Layer

CAM uses GPFS-SNC [92] to provide its storage layer. GPFS-SNC is designed as a cloud storage platform, which supports timely and resource-efficient deployment of VMs. GPFS-SNC manages the local disks directly attached to a cluster of commodity physical machines. More specifically, it has a number of unique features that make it a cloud-friendly storage system. First, GPFS-SNC supports co-locating all blocks of a file at one location, rather than stripping the file across machines. This enables a VM I/O request to be serviced locally from the stored location instead of remotely from physical hosts across the network. CAM

leverages this feature to ensure that co-located VM images are stored at one location and can be accessed efficiently. Second, GPFS-SNC supports an efficient block-level pipelined replication scheme, which guarantees fast distributed recovery and high I/O throughput through fast parallel reads. This feature is useful for CAM for achieving efficient failure recovery. Finally, GPFS-SNC specifies a user-level API that can be used to query the physical location of files. CAM uses this API to determine actual block location, and uses this information to infer storage closeness for data and VM placement.

3.1.2 Topology Awareness

MapReduce task scheduler uses the topology information of the cluster nodes to decide task assignments. The information is supplied by the user as a part of the job configuration file when the job is submitted. However, in an attempt to abstract hardware level details and present a simple interface to the user, existing cloud implementations do not expose the information about the topology of the cluster or the actual placement of VMs to the MapReduce scheduler [14]. Furthermore, the initial configuration provided by the user may become stale when the VMs are moved later.

CAM addresses these issues with three main components that together provide topology awareness as shown in Figure 3.1. First, the *CAM topology server* provides the additional topology information required to enable the MapReduce scheduler to place the tasks optimally. The topology server is an integral component of the ISAAC cloud service infrastructure and provides a REST interface, which the scheduler invokes. The information exposed by the topology server consists of network and storage topologies, and other dynamic node-level information such as CPU load. Second, a set of agents running on the physical nodes of the cluster periodically collect and convey to the topology server, a variety of pieces of data about the respective node, such as utilization of outbound/inbound network bandwidth, IO utilization and CPU/memory/storage load. The topology server consolidates the dynamic information it receives from the agents and serves it along with topology information about each job running in the cluster. The topology information is derived from existing VM placement configuration. Third, a new MapReduce task scheduler interfaces with the topology server to obtain accurate and current topology information. The scheduler readjusts task placement accordingly whenever a change in the configuration is observed. Note that CAM needs to provide a different scheduler because the standard MapReduce scheduler is designed to make the placement decisions only based on a static

configuration file and only at the beginning of the job [94]. While the MapReduce task scheduler is modified to leverage the topology and physical host resource utilization information in CAM, the MapReduce applications can run without any modification.

The **network topology** information is represented by a distance matrix that encodes the distance between each pair of VMs as cross-rack, cross-node, or cross-VM. Current MapReduce task schedulers consider rack and node localities but lack the notion of VM locality. When two VMs are placed on the same node, they are connected through a virtual network connection implemented as a part of the hypervisor. By virtue of the fact that the VMs share the same node hardware, the virtual network provides a high-speed medium that is significantly faster than the inter-node or inter-rack links. The network traffic between the VMs on the same node does not have to pass through the external hardware link. The network virtual device simply forwards the traffic in-memory through highly optimized ring buffers. CAM extends the MapReduce scheduler to consider this fine-grain locality information to make optimal placement choices for the tasks.

API	Description
`int get_VM_distance(string vm1, string vm2)`	Returns the distance between two VMs.
`struct block_location get_block_location(string src, long offset, long length)`	Returns the actual location of blocks.
`int get_vm_networkinfo(string VM, struct networkinfo)`	Returns the network utilization information of physical host on which the VM is running.
`int get_vm_diskinfo(string vm, string device, struct diskinfo)`	Returns disk utilization information.
`int get_VM_cpuinfo(string vm, struct cpuinfo)`	Returns CPU utilization information of the physcial host on which the VM is running.

Table 3.1 The key APIs provided by CAM to the MapReduce scheduler.

The **storage topology** information is provided as a mapping between each virtual device containing the dataset and the VM to which the device is local. In the native hardware context, a SATA disk attached to a node can be directly accessed through the PCI bus. In the cloud, however, the physical blocks belonging to a VM image attached to a VM could be located on a different node. Even though a virtual device might appear to be directly connected to the VM, the image file backing the device could be across the network, and

potentially closer to another VM in the cluster than the one it is directly attached to in the virtual setup. The topology server queries the physical image location through the GPFS API and presents the information to the MapReduce scheduler.

The specific APIs provided by the topology server is described in Table 3.1. *get_VM_distance*, provides MapReduce task scheduler with hints of the network distance between two VMs. The distance is estimated based on observed data transfer rates between the VMs, and is expressed in units of bandwidth. *get_block_location*, enables MapReduce to get the actual block location instead of the location of a VM, thus guaranteeing data locality. The rest of the calls are used to facilitate the MapReduce task scheduler to query the I/O and CPU contention information related to network and disk utilization. The MapReduce task scheduler can leverage this additional information to make smarter decisions, such as placing I/O intensive tasks on physical hosts that have idle I/O resources.

3.2 CAM Usage Model

CAM is a cloud platform with specific interfaces and support for running MapReduce jobs. The dataset to be processed is initially placed on GPFS. This is in contrast to most cloud models, which segregate storage and compute resources, and require the dataset to be moved from the storage cloud to the compute cloud for processing. Co-locating storage and compute clusters avoids the expensive ingestion phase for each job run. Data placed in this manner can be used by each subsequent job in CAM.

The placement of data is driven by the nature of MapReduce jobs that are typically run on the data. For instance, if the dataset is primarily used as input for various pattern search jobs, data locality is likely to be more important than task locality. The user can specify the nature of expected workloads, or the workload characteristics can be automatically derived based on previously observed I/O patterns.

The user submits a MapReduce job by providing the application, e.g., relevant java class files, indicating a previously uploaded dataset corresponding to the job, and the number and type of VMs to be used for the job. Each VM typically supports several MapReduce task slots depending on the number of virtual CPUs and virtual RAM allocated to the VM. The more the number of VMs assigned to a job, the quicker the job finishes.

CAM determines an optimal placement for the set of new VMs requested by the user by

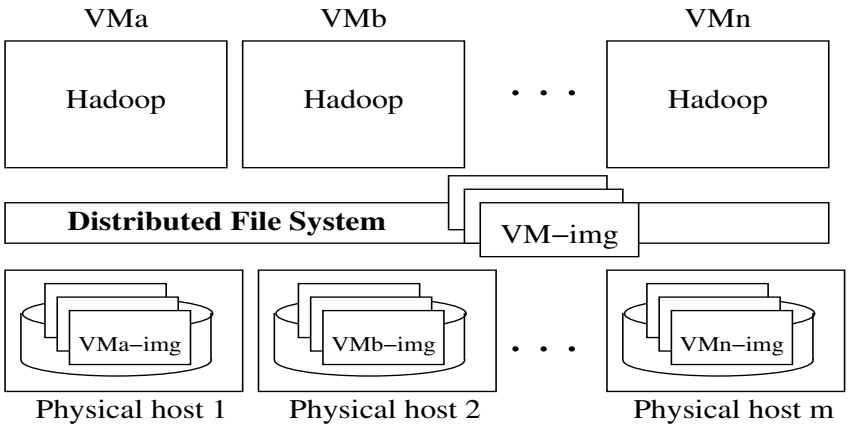

Figure 3.2 Setup of CAM for supporting MapReduce in the cloud.

considering a variety of factors such as current workload distribution among the cluster nodes, distribution of the input dataset required by the job, and the physical locations of the required master VM images. The images required to boot the VMs on the selected nodes are created from the respective master images using a copy-on-write mechanism provided by GPFS, which allows fast provisioning of a VM image instance without requiring a data copy of the master image. The job class files are copied into the cloned VM image by mounting the image as a loopback file system. These changes are private to the cloned image. Next, the data images are attached to the VMs and the respective device files are mounted within the VM for the MapReduce tasks to access the data contained within them.

Figure 3.2 illustrates the setup. Each machine is equipped with local disks. There is a distributed file system installed on top of these physical machines. The VM image files are stored in the distributed file system. Moreover, there is a cloud manager that allocates the resources for MapReduce jobs, and manages the data placement and VM placement.

3.3 Min-Cost Flow Based Placement

In this Section, we present how CAM manages Data and VM placement using a min-cost flow based approach. In our model, we assume that it is possible for the cloud provider to profile a job and estimate its characteristics such as job type (Map-Reduce intensive, Map intensive, or Reduce intensive), and input, output and intermediate data sizes. For our current implementation, we rely on user-provided or predetermined job descriptions to identify a job's type. However, the system can be easily extended to determine the amount

of time an application spends in different phases (Map, Reduce), and use this information to determine a job's type. For example, a job that spends more than 30% of the time in Map can be considered as Map-intensive.

3.3.1 Data Placement

We express the problem of optimally placing data in a given cloud cluster architecture as an instance of the well-known min-cost flow problem [39]. To achieve this, we break down the placement problem into three sub-problems, namely guaranteeing VM closeness, avoiding hotspots, and balancing physical storage utilization according to different job types. We capture the three constraints via similarly named factors in our model. *VM closeness* expresses how close data should be placed to VMs so that the network traffic between the corresponding VMs is minimized. *Hotspot factor* expresses the expected load on a machine, and identifies machines that do not have enough computational resource to support the VM(s) assigned to them. To avoid a hotspot, data needs to be placed on the least-loaded machine. This can be determined by measuring the current computational resource load of the machine and adding it to the expected computational requirements of the VMs that will work with the data to be placed on the machine. *Storage utilization* expresses the percentage of total physical machine storage space that is in use.

Job Type	VM closeness	Hotspot factor	Storage utilization
MR-intensive	Yes	Yes	Yes
M-intensive	No	Yes	Yes
R-intensive	No	No	Yes

Table 3.2 Significance of considered cost factors for different job types.

Table 3.2 shows the significance of the three factors on the performance of different MapReduce workloads. For workloads that are both Map and Reduce intensive, related data should be placed close together and on the least loaded machine. For Map intensive workloads, the data should be placed on the least loaded machine, but does not necessarily need to be placed close together due to the light shuffle traffic in such workloads. For Reduce intensive workloads, the only concern is the storage utilization of the machine on which the VM is to be placed. For all types of workloads, it is desirable to place data evenly across racks to

minimize the need to rearrange data over time for supporting migrating VMs.

We use these factors in constructing a min-cost flow graph that encodes the factors. Then we employ an extended solver to minimize the global cost of the graph, thus solving the original problem of determining how data should be placed in the virtualized cloud.

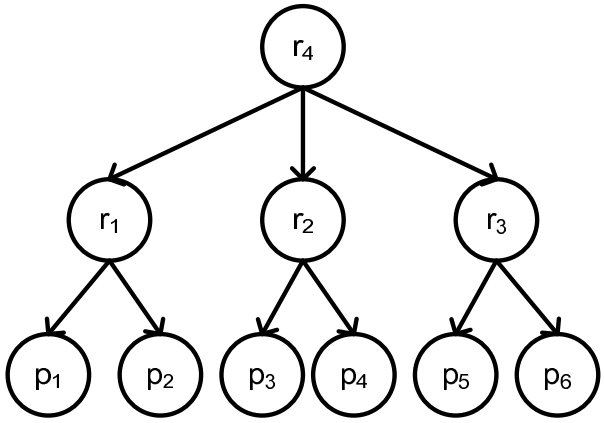

Figure 3.3 Sample network topology for data placement.

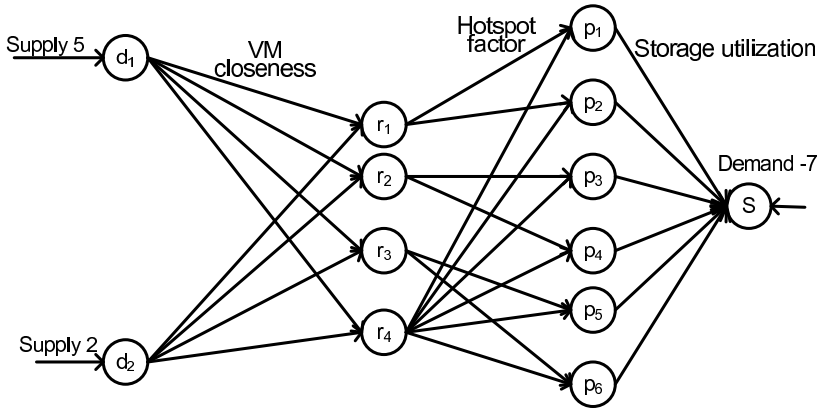

Figure 3.4 Flow graph for sample data placement.

Figure 3.3 shows a sample network topology, which consists of six physical nodes ($p_1, \ldots, p_6$) organized into three racks (r_1, r_2, r_3) with one master rack/switch (r_4) connecting the racks. Note that our model can support any topology where the network traffic can be estimated. There are several challenges when min-cost flow is used in our problem space. First, the three factors described above have to be encoded into the graph. Second, the correlation between different VMs images placement has to be encoded (which is shown to be non-trivial [39]). The flow-network model is aimed at minimizing the flow cost, however, we

employ the model to also consider VM closeness as an objective, which requires it to solve correlated constraints, i.e., a set of VMs would have to be placed together, but it does not matter where. Third, the three factors capture different costs that are not directly comparable to each other. For instance, *VM closeness* of 1 may signify the cost of copying 1 GB of data within a local rack, where as *Hotspot factor* of 1 may signify the cost of using a physical machine that has 1% more load than the least-loaded machine in the cluster. The two costs are clearly not the same. Thus, we need a way to formulate the three factors in the same units for encoding them into a min-flow graph.

In contrast to the extant data placement techniques that work at the granularity of the data blocks, our unit of data placement is a VM image. Such coarse placement is justified in CAM as the goal is to ensure that an entire image is available at one location. Moreover, our underlying storage layer of GPFS-SNC avoids striping the data across different physical machines, thus making block-level placement unnecessary.

We address these challenges as follows. Consider the corresponding min-cost flow graph for Figure 3.3 as shown in Figure 3.4. Here, two data items d_1 and d_2 with requests for 5 and 2 VM images, respectively, are submitted to the cloud. The number of VM images requested by a data item is denoted as the data item's *supply* for our flow graph. Conversely, we add a sink node S to the graph, that can "support" the VMs. The number of VMs that a sink node can handle is assigned as a *demand* value. In our example, S has a demand -7 and is the only place that can receive all the flows. Each flow graph edge has two parameters attached to it, the capacity of the edge and the cost for a flow to go through the edge. The data nodes, represented by d_1 and d_2 in the graph, have outgoing links to each rack with *VM closeness* as costs. The *Hotspot factor* is encoded in the links from the racks to each physical node p within its range. Note that even though r_4 serves as a switch between the racks, it is shown in the graph as directly connected to all the physical nodes. This is to ensure that the least-loaded machine can be chosen for Map-intensive jobs without being constrained by the network topology. All the physical nodes, $p_1, \ldots, p_6$, are linked to the sink node with *Storage utilization* as link costs. Note that there is no direct link from data item node d_j to the associated physical host p_i. This is to support scaling up the system, as otherwise the number of links in the graph will increase with increasing number of data items and physical nodes (much faster than the number of racks). Consequently, making it inefficient to solve for min-flow on the graph.

Table 3.3 provides the details of how we encode the various factors and system information in our min-cost flow graph. N_{d_j} is the number of VM images requested by dataset d_j. α_{jk}

	Data set d_j	Rack r_k	Physical host p_i	Sink S
Supply	$\sum(N_{d_j})$	0	0	-$\sum(N_{d_j})$
Incoming link from	N/A	N_{d_j}	Rack	Physical host
Outgoing link to (cap., cost)	Rack (N_{d_j}, α_{jk})	Physical host (Cap_i, β_i)	Sink (Cap_i, γ_i)	N/A

Table 3.3 Values assigned to the flow graph for data placement used in CAM.

captures *VM closeness*. The cost, α_{jk}, of outgoing link from the dataset d_j to physical host p_i on which the data is placed on rack r_k is estimated conservatively by the traffic in the shuffle phase as follows:

$$\alpha_{jk} = size_{intermediate} * \frac{num_{Reducer} - 1}{num_{Reducer}} * distance_{max}, \tag{3.1}$$

where $distance_{max}$ is the maximum network distance between any two nodes in the rack r_k, and $size_{intermediate}$ and $num_{Reducer}$ are the total size of data output by the Map phase and the number of reducers, respectively, of the MapReduce job running on data set d_j . Note that given its higher $distance_{max}$ a higher level rack/switch, e.g., r_4 in our example, would have a higher α than the lower racks, e.g., r_1, r_2 and r_3, based on this formula. The *Hotspot factor* is captured using β_i for physical node p_i, and is estimated by the current and expected load as follows:

$$\beta_i = a * (load_{exp} + load_{curr} - load_{min}), \tag{3.2}$$

where $load_{curr}$ and $load_{min}$ represent the current load and minimum current load, respectively. a is a parameter that acts as a knob to tune the weight of the *Hotspot factor* with respect to other costs. Moreover, based on guidelines from [35] the expected load is determined as $load_{exp} = \sum_j(\rho_j/(1-\rho_j) * CRes(d_j)$, where $\rho_j = \lambda_j/\mu_j$. Here, λ_j represents the number of d_j's associated jobs that arrive within a give time interval, μ_j represents the mean time for each VM to process a block, and $CRes(d_j)$ represents the compute resources required by jobs running on data set d_j. *Storage utilization* of a physical node p_i is captured by γ_i, which is determined by the current storage utilization compared with minimum storage

utilization of all p_is.

$$\gamma_i = b * (storageUtil_{p_i} - storageUtil_{min}) \tag{3.3}$$

Here, b is another parameter used to fine tune the weight of *Storage utilization* with respect to the other two factors. Finally, $Cap_i = freespace_{p_i}/size_{VMImg}$, is a conservative estimation of the capacity of each physical host calculated as the ratio of the available storage capacity of p_i and the size of the VM image. We assume that all VM images have the same size (10 GB in our experiments) when initially uploaded to the cloud.

To enable the graph to capture the correlation between VM image placement for one data request, we extend the solver to take into account an additional parameter for each edge, *split factor*, which specifies whether flows from a node are allowed to be split across different links, and is either *true* or *false*. In our example, *split factor* for all the links from d_1 and d_2 are set to *false*. This implies that all the flows from data nodes will wholly go through one of the $r_1, \ldots, r_4$, but will not be split between the racks.

Once a new data upload request comes in, the cloud server updates the graph and computes a global optimal solution. The graph is updated as follows. First, the graph is cleaned of data and state from the previous iteration. This is done by deleting the data nodes that correspond to the datasets that have finished uploading, and their outgoing links from the graph. Next, the cost of the edges corresponding to the *Hotspot factor* and *Storage utilization* of the physical nodes where the data was stored need to be updated using equations 3.2 and 3.3. Then, a new data node d_j is created for the new data upload request with edges to each rack node r_k with costs calculated based on the above equation 3.1. Once the graph is updated, a new min-flow value is calculated, which is then used by the cloud scheduler. This process ensures that the cloud scheduler is timely provided with updated information to accommodate varying loads.

3.3.2 VM Placement

The goal of VM placement is to maximize the global data locality and job throughput. Our model considers both VM migration and delayed scheduling of a job as part of the optimal solution. Delaying a job is used to explore better data locality opportunities that can arise in the near future, while minimizing time wasted during the waiting. Migrating a VM be-

longing to a job enables our scheduler to make room for other suitable jobs or to explore better location opportunity. There are two assumptions that we make about how VMs are migrated. First, we assume that once the VMs for a job are allocated, the job will not be suspended or killed. There is no preemption, which guarantees that the job will have some quota of resources at all times during its life span. Second, even if some of the VMs belonging to a job get migrated, their total number remains the same. We model the VM placement as minimum cost flow problem, which has similar characteristics to the min-cost flow based data placement.

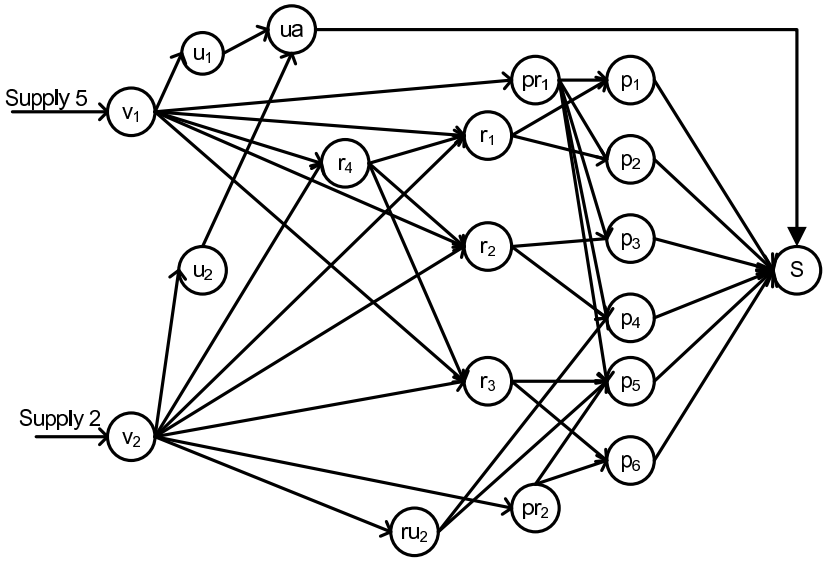

Figure 3.5 Flow graph for VM placement.

An example graph for VM placement is shown in Figure 3.5. Each job v_j is submitted to the system at the source node with the number of requested VMs, N_{v_j}, as the value of supply. The goal of the VM allocator is to either keep the job unscheduled (allocate 0) or allocate N_{v_j} VMs for each request. There is a single sink node, S, in the system with demand equal to minus the sum of the supply. The request from each job acts as a flow that goes either through the rack nodes, r_k, or through the unscheduled nodes, u_j, and finally to the sink. If a job is unscheduled, none of its VMs are allocated. Otherwise, the flow goes through the physical nodes, p_i. Each job v_j has a "preferred" node pr_j that has outgoing links to a set of physical hosts that would be preferable for v_j to be scheduled on. Based on the min-cost solution, a allocation scheme with min-cost can be easily derived. If the VMs are allocated to the highest level rack, it implies that the VMs can be allocated arbitrarily to any set of nodes in the VMs under the rack. Once v_j is scheduled, a "running" node (ru_j) is added to

the graph to keep track of information about the execution of v_j, which is then used by the solver to direct migration decisions.

The job type information is modeled as the cost of the edge from each job to the rack nodes in our flow based graph. The higher level rack has higher cost than the lower lever rack in terms of reduce traffic. We use conservative approximation to compute bounds on data transfer costs. The cost to the highest level rack is estimated by worst case VM arrangement with regards to the map and reduce traffic. Similar rules apply to the lower level rack. The cost of the edges to the unscheduled nodes are set to be increased over time so that delayed jobs get allocated sooner than recently submitted jobs. This cost also controls when a job stops waiting for better locality, and thus offers a knob to tune the trade-off between data locality and latency. The aggregated unscheduled nodes control how many VMs can remain unscheduled, which is another system parameter to control the system resource utilization and data locality trade-off. The cost of the edges to running nodes set is increased over time and is job-progress aware. For example, a reduce intensive job run during the reduce phase might not be suitable for migration to make room for a contending job request.

Similarly as for data placement, we provide means for expressing the cost of reading data across different level of racks, migrating VMs, and delay scheduling in the same units. For example, we can choose that the copying of 1 GB data across rack local switch costs the same as copying 0.5 GB data across one level higher rack, or the same as setting up of and starting one VM, or the same as delaying a VM execution by say 10 seconds.

	Job node set	Preferred node set	Running node set	Unscheduled node set	Unscheduled aggregator node	Rack node set	Physical host node set	Sink
supply	$\sum(N_{v_j})$	0	0	0	0	0	0	$-\sum(N_{v_j})$
Incoming link from	N/a	job	job	job	all unschedule nodes	job; higher rack	rack; preferred nodes set; running nodes set	physical host; unscheduled aggregator
Outgoing link to (cap., cost)	Rack(N_{v_j}, ρ_j) Prefer(N_{v_j}, θ_j) Run(N_{v_j}, ϕ_j) U(N_{v_j}, ϵ_j)	Physical host $(d_i, 0)$	Physical host $(r_i, 0)$	Unscheduled aggregator $(N_{v_j}, 0)$	Sink $(N_{unsched}, 0)$	Physical host $(N_{r_k}, 0)$	Sink $(N_{vm}, 0)$	N/A
flow	N_{v_j}	$0/N_{v_j}$	$0/N_{v_j}$	$0/N_{v_j}$	$0/N_{unsched}$	$0, N_{r_k}$	$0, 1$	$\sum(N_{v_j})$

Table 3.4 Values assigned to the flow graph for VM placement used in CAM.

We categorize the various nodes in the graph into different types as shown in Table 3.4.

- Preferred node set (pr_j): These graph nodes point to a set of physical nodes p_i that have a job v_j's associated dataset stored on them. An edge from a preferred node to p_i has the cost of 0 and the capacity of the number of VM disk images stored on p_i.

- Running node set (ru_j): These are dynamically added nodes that point to p_is that are currently hosting the v_j's VMs. An edge from ru_j to p_i has a cost of 0 and the capacity of the number of VMs running on p_i.

- Unscheduled node set (u_j): These nodes provide information about currently unscheduled jobs. u_j has an outgoing edge with capacity of N_{v_j} and cost 0 to a unscheduled aggregator.

- Unscheduled aggregator node (ua): The graph contains a single unscheduled aggregator. u has an outgoing edge with cost 0 to the sink with capacity of $N_{unsched} = \sum(N_{v_j}) - M + M_{idle}$, where M is the total number of VMs that the cluster can support and M_{idle} denotes the number of idle VM slots allowed in the cluster.

- Rack node set (r_k): The rack node r_k represents a rack in the topology of the cluster. It has outgoing links with cost 0 to its subracks or, if it is at the lowest level, to physical nodes. The links have capacity N_{r_k} that is the total number of VM slots that can be serviced by its underlying nodes.

- Physical host node set (p_i): Each physical host p_i has an outgoing link to the sink with capacity the number of VMs that can be accommodated on the physical host N_{vm} and cost 0.

- Sink S: The single sink node with demand $-\sum(N_{v_j})$.

- Job node set (v_j): This set represents each job node v_j with supply N_{V_j}. It has multiple outgoing edges corresponding to the potential VM allocation decisions for v_j. These edges are discussed in the following:

  - Rack node set r_k: An edge to r_k indicates that r_k can accommodate v_j. The cost of the edge is ρ_j that is calculated by the map and reduce traffic cost. If the capacity of the edge is greater than N_{v_j}, it implies that the VMs of v_j will be allocated on some p_is on the rack.
  - Preferred node set (pr_j): An edge from job v_j to the job wide preferred nodes set pr_j has capacity N_{v_j} and cost θ_j. The cost is estimated by only the reduce phase traffic, because in this case map traffic is assumed to be 0.
  - Running node set (ru_j): A link from job v_j has capacity of N_{v_j} and cost $\phi_j = c*T$, where T is the time the job has been executing on the set of machines and c is a constant used to adjust the cost relative to other costs.

– Unscheduled node set (u_j): An edge to the job-wide unscheduled node u_j has capacity N_{v_j} and cost ϵ_j, which corresponds to the penalty of leaving job v_j unscheduled. $\epsilon_j = d * T$, where T is the time that job v_j is left unscheduled and d is a constant used to adjust the cost relative to other costs. The *split factor* for this link is marked as `true`, which means the allocation of all the VMs are either satisfied or be delayed until the next round.

When a VM allocation request is submitted, the flow graph is updated to calculate a new global optimal solution for the VM scheduler. Similar to the update process for data placement, the graph is cleaned by removing unnecessary nodes and edges. For example, for each finished job v_j, the associated nodes including the unscheduled node u_j, the preferred node pr_j and the running node ru_j are deleted from the graph since the job has released its VM resources. Then, the costs of edges related to the jobs that are still running are updated according to Table 3.4 to reflect the jobs' current state. Next, a set of new nodes and edges are added into the graph for the current VM allocation request, namely, a job node, a related unscheduled node, and a preferred node. Moreover, the corresponding edge costs are again calculated as described in Table 3.4.

Once the solver outputs a min-cost flow solution, the VM allocation assignment can be obtained from the graph by locating where the associated flow leads to for each VM request v_j. Flow to an unscheduled node indicates that the VM request is skipped for the current round. If the flow leads to a preferred nodes set, the VM request is scheduled on that set of nodes. Finally, if the flow goes to a rack node, it implies that the VMs from the job are assigned to arbitrary hosts in that rack.

The number of flows sent to a physical host through rack nodes or preferred nodes set is not higher than the number of available VMs of each physical hosts. This is guaranteed by the specified link capacity from physical host to sink. Thus, all VM requests that are allocated will be matched to a corresponding physical host.

3.4 Evaluation

In this section we show the effectiveness of our approach through a set of Hive [95] based, I/O-bound micro-benchmarks running on a real cluster. We evaluate CAM's network and storage topology awareness against vanilla Hadoop, as to our best knowledge, CAM is the

first technique that reintroduces the concept of data locality by exposing topology information in a cloud setting. We also compare CAM's data and VM placement against a state-of-the-art technique using a mix of workloads on a large simulated cluster.

3.4.1 Micro-Benchmark Results

In this section, we use an I/O intensive workload based on a Hive benchmark to show the effectiveness of topology awareness and storage awareness for task placement. The reported numbers are averages across three runs of a test.

Our cluster consists of 4 RHEL 6.0 physical machines that use KVM as the hypervisor. Each machine has two quad-core 2.4 GHz Intel E5620 processors and 48 GB of main memory. The machines are organized in one rack and are connected to a dedicated Gigabit switch. We launched 23 VMs, 1 master and 22 slaves, on the four physical hosts. Each VM is configured with 1 GB of main memory and two 2.4 GHz vCPUs. Each VM has one map slot and one reduce slot, with a map block size of 64 MB. MapReduce fair scheduler is employed.

Number of jobs	21	9	7	4	3	3	3
Map tasks / job	1	2	10	50	100	200	400

Table 3.5 Distribution of job sizes in terms of number of map tasks used for micro-benchmark tests.

We generate a job submission schedule with 50 I/O-intensive Grep jobs using Poisson distribution with job inter-arrival time 10 seconds. To make the comparison consistent, we generate the submission schedule with a submission duration of 554 seconds, record it into a file and use it throughout the experiments. The size distribution of each Grep job for this experiment is shown in Table 3.5 and is based on the experiments performed by Zaharia et al. [41]. Thus, our schedule is representative of a typical workload of a production MapReduce cluster with a mix of many small jobs with a single map task per job, and a few large jobs with more than 100 map tasks per job. The input for the Grep jobs is generated using Teragen [96], with each map file consisting of 100 M records of size $0.1K$ for a total input size of 10 MB.

3.4.1.1 Impact of Network Topology Awareness

In our first experiment, we use our submission schedule to evaluate the impact of network topology awareness in a CAM-based implementation of Hadoop [96]. For this purpose, we measure the execution time for our schedule. We also measure the achieved locality expressed as the percentage of total Map tasks that are scheduled on the VMs (for VM locality) or physical nodes (for node locality) that have the associated data. As a base case for comparison we use vanilla Hadoop, which is unaware of the actual network topology and in this case cannot determine if two VMs are running on the same node or not.

System	VM locality	Node locality	Average execution time
Hadoop	29.1%	42.6%	48.3 s
CAM-based Hadoop	29.0%	49%	34.2 s

Table 3.6 Impact of network topology awareness on Hadoop performance.

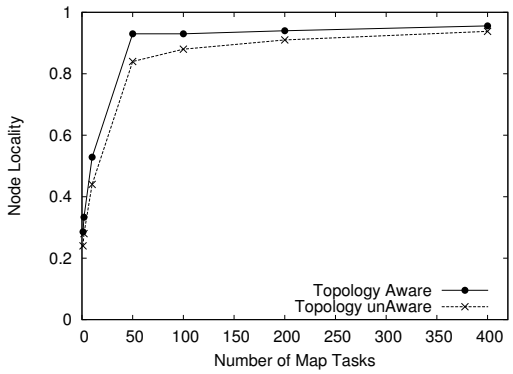

Figure 3.6 Breakdown of observed locality for jobs with different number of map tasks, with and without network topology awareness.

The results are shown in Table 3.6. We observe that by exposing topology information to Hadoop, the node locality is improved by 6.4%, and the average job execution time reduces by 8%. Figure 3.6 shows a break-up of the node locality in terms of the number of map tasks for the two studied cases. Observe that network topology information effectively improves the node locality for jobs with 10 and 50 map tasks by 8% and 9%, respectively. Jobs with more than 50 map tasks see a decreasing improvement, because with the increased number of maps in the small cluster the chance of co-locating map tasks on the same node also

increases. However, topology awareness is important even in such a small cluster as most MapReduce jobs have fewer than 50 map tasks [41]. Note that the relatively good performance of vanilla Hadoop is due to the fact that the test cluster consists of a small number of physical hosts located on the same rack.

3.4.1.2 Impact of Storage Topology Awareness

System	Average execution time
Hadoop	65.6 s
CAM-based Hadoop	48.3 s

Table 3.7 Impact of storage topology awareness on Hadoop performance.

In our next experiment, we observe the impact of providing storage topology hints to Hadoop. For this test, we use the 22 VM slaves with local data, and then migrate 6 of the VM images from one physical host to another. This makes 27% (= 6/22) of the data to become remote. Once again we measure the average job execution time for our schedule. The results are sown in Table 3.7. We observe that storage awareness can help improve the MapReduce execution time for our job schedule by 26.5%, on average.

These results show that CAM-based Hadoop can provide better performance for Hadoop tasks by exposing network and storage topology information to the Hadoop scheduler.

3.4.2 Macro-Benchmark Results

Number of jobs	38	16	14	8	6	6	4	8
Map tasks / job	1	2	10	50	100	200	400	800

Table 3.8 Distribution of job sizes in terms of number of map tasks used for macro-benchmark tests.

In our next set of experiments, we show the effectiveness of our approach. For this purpose, we extend the simulator PurSim [35] to include the min-cost flow data placement, VM placement, network awareness, and storage awareness mechanisms described in Section 3.3. PurSim is a network flow level discrete event simulator that simulates the MapReduce ex-

ecution semantics. Similarly as in previous tests, we generate a schedule with job size distribution based on Zaharia et al. [41] shown in Table 3.8. For these experiments the interarrival time is randomly generated between 60 and 90 seconds.

3.4.2.1 Data and VM Placement

In this section, we evaluate the effectiveness of min-cost flow (MCF) Data and VM placements used in CAM. We consider three types of MapReduce workloads, namely Map-intensive, MapReduce-intensive, and a workload with a mix of Map, MapReduce, and CPU-intensive jobs.

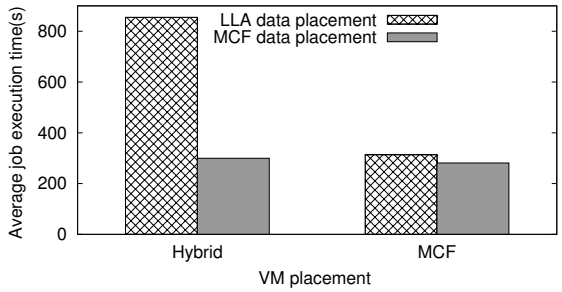

Figure 3.7 Execution time for Map-intensive workloads.

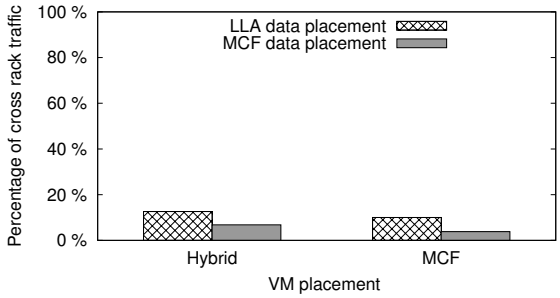

Figure 3.8 Fraction of data accessed remotely for Map-intensive workloads.

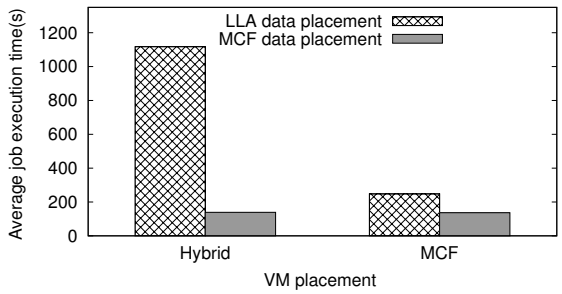

Figure 3.9 Execution time for MapReduce-intensive workloads.

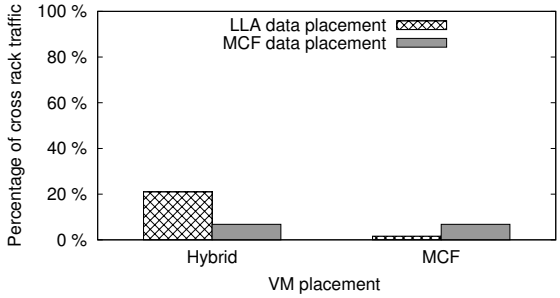

Figure 3.10 Fraction of data accessed remotely for MapReduce-intensive workloads.

We consider two data placement strategies, namely load and locality aware (LLA) data placement and MCF data placement. We also consider two VM placement strategies, namely Hybrid VM placement and MCF VM placement. The LLA and Hybrid strategies are defined in Purlieus, and are used as a baseline for comparison to CAM's MCF based approach.

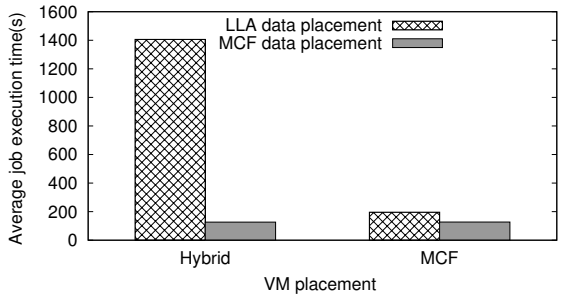

Figure 3.11 Execution time for Mixed workloads.

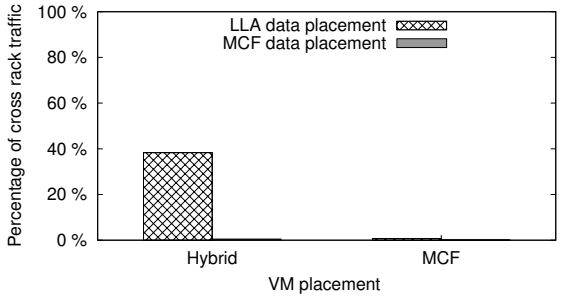

Figure 3.12 Fraction of data accessed remotely for Mixed workloads.

Figures 3.7, 3.9, and 3.11 show the average execution time for the three workloads considered. Similarly, Figures 3.8, 3.10, and 3.12 plot the percentage of total data (for the tests) that is accessed remotely across the rack under each combination of VM and data placements for the three studied workloads. For the Map-intensive workload, the combination of MCF VM and data placement produces a $3x$ speedup over the baseline, which is due to a $3.3x$ decrease in relative cross rack traffic. On the other hand, the same combination for the MapReduce-intensive traffic produces an $8x$ speedup with a corresponding 3 fold decrease in network traffic. The MCF placements see the best speedup of $8.6x$ verses the baseline for the mixed workload, due to the fact that they all but eliminate cross network traffic.

For all workload types using either the MCF data placement combined with the baseline VM placement, or conversely using the MCF VM placement with the baseline data placement, produces a significant speedup. Hence, the MCF graphs constructed for both placement optimization problems successfully optimize the respective factors and produce an optimal solution. Note that combining both techniques does not yield further significant benefit.

3.4.2.2 Impact of Network Topology Awareness

Network topology awareness	VM locality	Average job execution time
Unaware	82%	24.6 s
Aware	99%	22.4 s

Table 3.9 Impact of network topology awareness on Hadoop performance.

In this experiment, we configure our VM cluster to run 100 jobs simultaneously on 192 VMs using Hadoop fair share scheduling mechanism. Table 3.9 shows that network topology

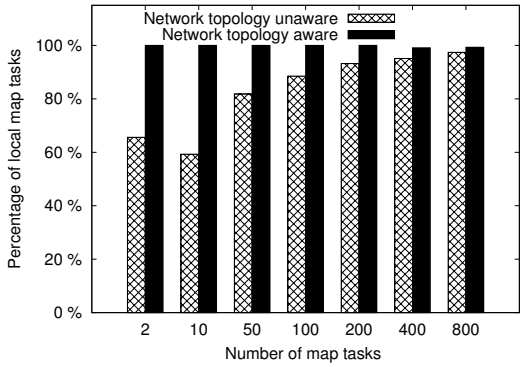

Figure 3.13 Impact of network topology awareness on locality of map tasks broken down in terms of number of map tasks.

awareness improves the map tasks locality on average by 7%, and reduces the average job execution time by 9%. Figure 3.13 shows the breaks up for the percentage of map VM locality with respect to the number of map tasks. We observe that network topology awareness is most effective for jobs that have less than 50 map tasks, improving locality by 24.2% on average.

3.4.2.3 Impact of Storage Topology Awareness

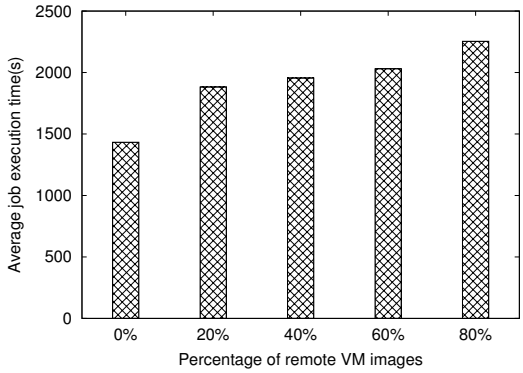

Figure 3.14 Impact of storage topology awareness on MapReduce performance in terms of percentage of remote VM images.

In our next experiment, we demonstrate the effectiveness of providing storage topology information hints to MapReduce. We vary the number of VM image files that are placed remotely with respect to the physical node where the VM is to be run. First, we measure how loss

of VM image locality affects the average execution time. Figure 3.14 shows the results. We observe that as more VMs are placed remotely, the average job execution time increases. For example, with 80% remote VM images, the average MapReduce job execution time worsens 36% compared to the all local images case (0%). As seen from the previous experiments, CAM-based Hadoop achieves all local images using the storage topology information, and thus offers an effective solution for VM placement.

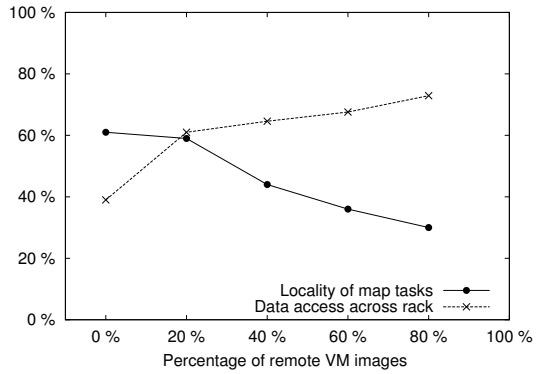

Figure 3.15 Impact of storage topology awareness on MapReduce performance.

Next, we measure the locality of map tasks achieved with varying VM images placed remotely from their physical host. Figure 3.15 shows the percentage of the number of local map tasks with varying remote VM images. We see that without exposing storage topology information, the locality of map tasks is decreased. Conversely, the amount of data accessed remotely across the rack increased. Thus, by exposing storage locality, CAM can minimize the cross rack traffic due to remotely accessing VM images.

3.4.3 Scalability of CAM

We now discuss the scalability of our min-cost flow model. In our macro-benchmark experiments we found the overhead to be negligible in a cluster of size 192 after repeated tests. As shown in Quincy [39], even for a large cluster size (thousands of nodes) a similarly-sized flow network can be solved in a few seconds, which is significantly smaller than the running time of typical MapReduce jobs. That is because techniques such as successive approximation push-relabel can process large-scale graphs efficiently. Moreover, the overhead of the solver is incurred only once, when the job is submitted for scheduling.

In summary, CAM offers to simultaneously meet the different constrains to co-locate VM and data on physical hosts, and improves overall MapReduce in the cloud application performance.

3.5 Chapter Summary

In this Chapter, we have presented the design of CAM, an innovative resource scheduler designed to address performance degradation of MapReduce jobs when running in virtualized clouds. CAM adopts a three level approach to avoid placement anomalies due to inefficient resource allocation: placing data within the cluster that run jobs that most commonly operate on the data; selecting the most appropriate physical nodes to place the set of virtual machines assigned to a job; and exposing, otherwise hidden, compute, storage and network topologies to the MapReduce job scheduler. CAM uses a flow-network-based algorithm that is able to reconcile resource allocation with a variety of other competing constraints such as storage utilization, changing CPU load and network link capacities. Evaluation of our approach using both micro-benchmarking and simulation on a 23 VM cluster shows that compared to a state-of-the-art resource allocator, CAM reduces network traffic and average MapReduce job execution time by a factor of 3 and 8.6, respectively.

Chapter 4

MapReduce Online Performance Tuning

As we mentioned in Chapter 1, efficient resource utilization relies on the partnership between cloud providers and users. We have introduced a min-cost flow resource manager for hosting MapReduce clusters in the cloud for service providers in Chapter 3. We then present a user level tool that helps enhance the accuracy of resource requests. The two together guarantee efficient resource management of MapReduce in the cloud.

Within MapReduce framework, developers specify required resources for their applications through parameter configuration. However, MapReduce job parameter tuning is a daunting and time consuming task. The parameter configuration space is huge. More than 70 parameters impact job performance. It is also difficult for users to figure out the optimal value without first having a clear understanding of the MapReduce application characteristics. The key challenge is to systematically explore the configuration space to determine a near-optimal configuration. Extant offline tuning approaches are slow and inefficient as they rely on multiple test runs and significant human effort.

In this Chapter, we propose an online performance tuning system, mrOnline, to improve MapReduce application performance. mrOnline monitors the job execution, tunes the parameters based on collected statistics and provides fine-grained control over parameter configuration changes. Particularly, our system allows each task to have a different configuration in parallel instead of using the same configuration for all the tasks. We design a gray-box based smart hill climbing algorithm that can effectively converge to a global optimal configuration with high probability.

mrOnline improves single Hadoop job performance via online performance tuning, as well as expedites the performance tuning process by reducing the number of test runs by employing a finer grain online process (multiple configurations per test run). mrOnline provides the

ability to tune multiple jobs performance in a multi-tenant environment. Moreover, mrOnline considers dynamic cluster utilization information to help MapReduce applications avoid hot spots. mrOnline does not require any modification to user programs, which makes it user friendly and encourages the adoption of it.

The rest of the chapter is organized as follows. Section 4.1 presents the introduction of YARN, the classification of configuration parameters and two use cases mrOnline focusing on. Section 4.2 discusses the system architecture of mrOnline after which comes the explanation of task level dynamic configuration. In Section 4.4, we detail the design of gray-box based hill climbing algorithm to systematically search for optimal configuration parameters followed by the description of tuning rules for various key job configuration parameters. Section 4.7 demonstrates the effectiveness of mrOnline compared against default configuration through a serial of experiments. Section 4.8 summarizes the chapter.

4.1 Background of mrOnline

In this section, we first describe enabling technologies for mrOnline, and then identify two specific use cases for which we have designed mrOnline.

4.1.1 YARN

mrOnline is designed and implemented on YARN [34], the latest generation of the publicly available Hadoop platform. We choose YARN as it provides many advantages over traditional Hadoop. Hadoop is designed as a monolithic framework, which tightly couples the MapReduce programming model with distributed resource management. This leads to misuse of the MapReduce programming model in order to just leverages the large scale compute resources provided by enterprises, research organization and other institutes. For instance,users submit map-only applications to launch any number of processes in the cluster [34] or submit applications which have map function implemented as reduce tasks when the map quota is limited [97]. Moreover, Hadoop employs a centralized scheduler for managing the tasks of all jobs. This is becoming a performance bottleneck as the number of jobs submitted to a Hadoop cluster grows. Traditional Hadoop implementation also does not support changing the Map or Reduce slot configurations between different jobs, which precludes dynamically adapting to variations during a job's life-cycle and reduces efficiency.

YARN has been designed to address these limitations in Hadoop. YARN separates the computational programming models from resource management, and supports frameworks other than MapReduce such as Giraph [98, 99], Spark [100, 101], and Storm [102]. In this paper, we focus on tuning parameters of the MapReduce programming model running on top of YARN. However, YARN provides for means to extend our approach to support performance tuning of other frameworks as well. Another useful feature of YARN is that it delegates application related scheduling to per-application *masters* than can use application specific resource scheduling, thus providing for higher scalability. For this purpose, YARN manages cluster-wide resources through a key concept, "containers." A container is a resource scheduling unit that specifies the number of CPUs, the memory needed, etc. Different MapReduce applications can request different size of containers from YARN as needed by the application. For example, an application master is responsible for specifying the number of needed containers, the size of each container, the mapping between the containers and tasks etc. mrOnline leverage this flexibility offered by containers to design a task-level configuration framework.

4.1.2 Parameter Classification

We focus on parameters that impact application performance and are suitable for dynamic configuration. The optimal values of these parameters depend on the application characteristics, the size and the content of input data and the cluster step. We classify performance related parameters into three categories based on when a modified parameter can become effective.

The first category includes parameters that are difficult to change after the application is started. The number of maps, the number of reducers, slow start (mapreduce.job.reduce.slowstart.completedmaps) are three key parameters that fall into this category. Slow start is a parameter that specifies how many maps should be completed before launching the first reduce tasks. Starting reduce tasks early helps overlap the map tasks execution with the shuffle phase. However, starting reduce tasks too early occupies the cluster resources needed by map tasks. The optimal value is application specific.

The second category consists of parameters that cannot be changed on the fly for already running tasks but impact the tasks that will be launched in the near future. Examples include io.sort.mb, the number of virtual cores in containers, the memory size of containers,

and parameters specifying reduce buffer size. Specifying the right value for this category of parameters can reduce disk I/Os and improve the cluster utilization.

The third category consists of parameters that can be changed on the fly and become effective immediately. For example, mapred.inmemmerge.threshold, io.sort.spill.percent fall into this category. These two parameters control the threshold of when to spill out data into disk. mrOnline can even try multiple values within a task for parameters in this category thus enhance the tuning efficiency.

mrOnline currently supports tuning of parameters in the second and third category. Tuning of parameters in the first category can be done using simulation tools, such as MRPerf [63]. Moreover, we also consider the mrOnline-enabled tuning of parameters in the first category in our ongoing research.

4.1.3 Use Cases of mrOnline

There are two use cases we considered when designing mrOnline. The first use case is to expedite test runs by trying out multiple task executions in a single test run. In this way, we can reduce the tuning process significantly. The second use case is to improve performance of applications that are executed only once. mrOnline employs different strategies for these two use cases.

Use Case One: Expediting test runs. In this use case, we aggressively and systematically search for different parameter configurations aimed at finding the optimal values. We first design and implement a task level configuration framework that enables testing of different parameter configurations in a single test run. We then propose a gray-box based smart hill climbing algorithm to find the optimal configuration. The quality of the generated solution depends on the number of tasks executed in a single test run. If too few tasks are executed, the configuration quality can be improved by multiple test runs. We also improve the algorithm convergence efficiency by monitoring and modeling the runtime statistics into the algorithm.

Use Case Two: Improve the application performance in a single run. For this use case, we conservatively tune the configurations mostly based on the observed runtime statistics. For example, if we observe extra spills are written to disk, we increase the sort buffer in map phase. Our main goal here is to improve the performance instead of searching for optimal configuration.

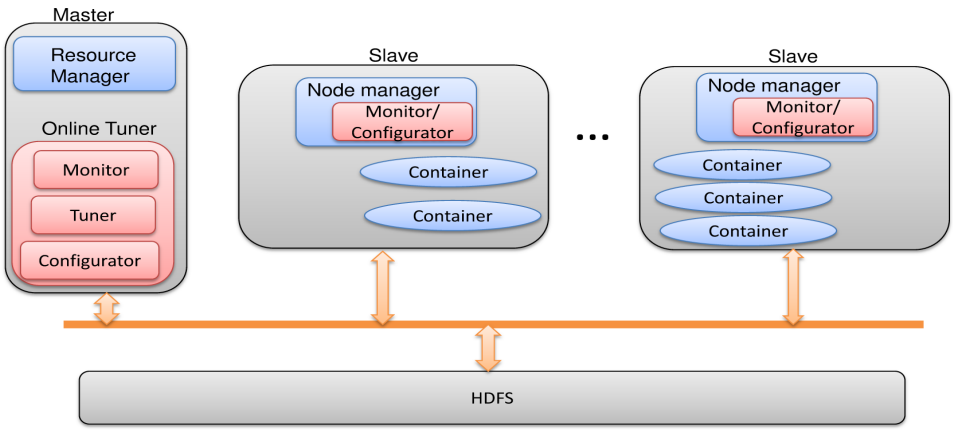

Figure 4.1 mrOnline System Architecture.

4.2 Architecture of mrOnline

The overall architecture of mrOnline is shown in Figure 4.1. mrOnline is based on YARN that, unlike Hadoop centralized job tracker, has a resource manager that manages cluster resources, the execution cycle of distributed application-specific masters, and tracks node liveness. To support dynamic configuration, mrOnline has to modify the YARN resource manager to support allocation of different-sized containers for different applications. The Node manager is akin to the task tracker of Hadoop, and runs on each slave node and is responsible for managing the containers running locally. However, YARN delegates the task tracking functions to per application components. mrOnline implements the sub-components within each node manager to leverage the existing features such as resource monitoring.

mrOnline consists of a centralized master component, *online tuner*, which is a daemon that can runs on the same machine as the resource manager of YARN or on a dedicated machine. Online tuner controls a number of distributed slave components that run within the node manager on the slave nodes of YARN cluster.

Online tuner is composed of three components: a *monitor*, a *tuner* and a *dynamic configurator*. The monitor works together with the slave monitors that run within node managers on each slave node to periodically monitor application statistics. More specifically, the slave monitors gather the statistics of tasks running locally and node statistics and send the information to the centralized monitor. The centralized monitor then aggregates the information, summarizes it if necessary and pass the information to the tuner.

The tuner implements the tuning strategies and algorithms, which decide what parameters should be changed and what new values should be assigned to each parameter for each ap-

plication task. It then generates the new configurations for each application and each task. Finally, the dynamic configurator takes the new configuration from the tuner and distributes the lists of new parameters to the slave configuration components. The slave configurators are responsible for activating the new changes for tasks running locally.

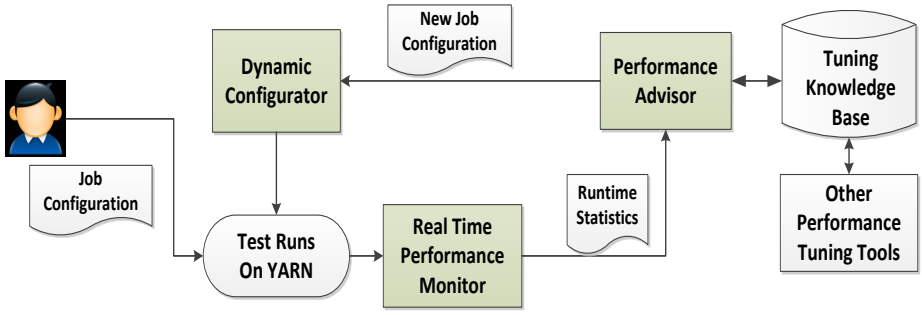

Figure 4.2 mrOnline tuning process.

Figure 4.2 illustrates the tuning process of mrOnline. After storing input datasets in HDFS, a user launches the application using a default configuration or a configuration based on rough understanding of application characteristic. The real time performance monitor monitors runtime statistics include per task information such as task progress rate, CPU, memory utilization, the number of spill records, I/O utilization and per node resource utilization information and sends them to performance advisor that is the tuner component in Figure 4.1. Performance advisor calculates new configurations and sends them to the dynamic configuration manager. The configuration manager then changes the configuration for each task correspondingly. The tuning process can iterate for multiple runs until a desirable configuration is generated. mrOnline supports aggressive and conservative strategies for the two use cases described in section 4.1. The performance tuning advisor can also be extended to communicate with other performance tuning tools, and the tuning knowledge can also be stored in a tuning knowledge base.

4.3 Task-level Dynamic Configuration

We extend YARN to support task level dynamic configuration. In contrast to traditional YARN applications that use a configuration throughout the task execution, mrOnline enables YARN applications to have different configurations for each tasks.

Within the YARN framework, when the dynamic configurator gets new lists of tasks, configurations mappings, it writes per task configuration files to the working directory of the corresponding application. When tasks are assigned to containers running on remote slave nodes, the slave configurator is responsible for downloading the changed configuration file for the launched tasks. The launched tasks then read the changed configuration files thus changing the configuration that might be different from the original job configuration file.

Current YARN supports the same container size for all map tasks or all reduce tasks only. We extend the resource scheduler to support requests that require different size of containers. More specifically, we extend the hash map data structure to keep track of the size of the requested containers and the corresponding operations.

API	Description
`List<String> getConfigurableJobParameters (JobID jid)`	Returns the set of configurable parameters for the job with job ID `jid` and associated tasks that are currently running or will run in the future.
`List<String> getConfigurableTaskParameters (JobID jid, TaskID tid)`	Returns the set of configurable parameters for the tasks with job and task IDs `jid` and `tid`, respectively.
`int setJobParameters (JobID jid, Map<String,String> kv)`	Sets the parameters for a job with ID `jid`.
`int setTaskParameters (JobID jid, TaskID tid, Map<String,String> kv)`	Sets the parameters for a task with job and task IDs `jid` and `tid`, respectively.
`int setTaskParameters (JobID jid, Map<String,String> kv)`	Sets the parameters for all the tasks associated with a job with ID of `jid`.

Table 4.1 Key APIs provided by the Dynamic Configurator of mrOnline.

The key APIs supported by dynamic configurator of mrOnline is described in Table 4.1. $getConfigurableJobParameters$ and $getConfigurableTaskParameters$ return the set of configurable parameters for a specified job or task. The other three APIs set the job or task configuration parameters with specified values. The APIs also enable other tuning algorithms to tune the job parameters easily if needed.

4.4 Gray Box based Hill Climbing

Algorithm 1 Gray Box based Hill Climbing.

```
1: Initialize LHS parameters k, m, n, the threshold of neighborhood size N_t, the shrink factor f
   and the threshold of global search g.
2: local_search = 1, global_search = 1
3: config[m] = LHS_sampling(m)
4: C_cur=best(config[m])
5: N_{c_cur}=adjust_neighbor(C_cur)
6: While global_search < g do
7:     if local_search == 1 then
8:         while N_{C_cur} > N_t do
9:             config[n]=LHS_sampling(n)
10:            C_candi=best(config[n])
11:            if(cost(C_candi) < cost(C_cur))
12:                C_cur=C_candi
13:                N_{c_cur}=adjust_neighbor(C_cur)
14:            else
15:                N_{c_cur}=shrink_neighbor(C_cur)
16:            endif
17:        endwhile
18:        local_search = 0
19:    endif
20:    config[m]=LHS_sampling(m)
21:    C_candi=best(config[m])
22:    if(cost(C_candi) < cost(C_cur))
23:        C_cur = C_candi
24:        N_{c_cur}=adjust_neighbor(C_cur)
25:        local_search = 1
26:    else
27:        global_search++
28:    endif
29: endwhile
```

In this section, we present the design of our tuner that systematically searches through the configuration space and finds a desirable configuration given a specific application, data set size and cluster configuration. More specifically, we introduce a gray-box based hill climbing algorithm to tune the job parameter configurations for YARN applications. Our gray-box based hill climbing is inspired by smart hill climbing algorithm [66]. Smart Hill Climbing is an algorithm that was developed to provide black-box optimization for web application server configuration. Our approach provides three desirable properties: 1) it supports probabilistic guarantees on how close the determined configuration to the optimal configuration;

2) it effectively tolerates the noise in evaluated cost from various factors such as resource contentions; 3) it adopts weighted latin hypercube sampling (LHS) technique that helps improve the sampling quality and increase the convergence speed. The intuition in applying LHS is to partition the probability distributions of each parameter into equal probability intervals and sample a value from each interval. In addition to challenges we face when applying the algorithm, we also consider monitored information to speed up the search process. Thus, our algorithm is a gray-box based method.

Algorithm 1 describes the detailed approach we have devised in mrOnline. Our algorithm has two phases, a global search and a local search phase. The global search phase aims to find the promising local area to explore through efficient space filling sampling strategy. The local search phase narrows down the search neighborhood size by recursively applying LHS search until the neighborhood size is smaller than a predefined threshold.

The algorithm parameters are first initialized including the number of LHS intervals k, the number of sampled configurations in global search phase m, the number of sampled configurations in local search phase n, the threshold of neighborhood size N_t and the shrink factor f. The LHS interval indicates the granularity of each parameter interval. In our evaluation, we use $k = 24$. Shrink factor f controls the ratio of the current neighborhood size to the shrunken neighborhood size. After the initialization, we enter the global search phase, by using LHS to generate m configurations. We then configure the first m tasks to use the generated m configurations. The monitor then periodically monitors the application performance that is then plugged into equation 4.1 to estimate the cost of each configuration (we explain how we determine y later). We choose the configuration C_{cur} that has lowest estimated cost as the current search point and set the neighborhood size based on C_{cur}. Next, we go to local search phase. It iteratively applies LHS sampling with n sampled configurations on the updated neighborhood with the center point C_{cur}. The dynamic configurator uses the newly calculated configurations to configure the newly launched tasks dynamically and the monitor component then gathers the execution statistics of launched tasks. A candidate configuration C_{candi} is chosen based on the updated minimum estimated cost. The algorithm compares the estimated cost of the candidate configuration C_{candi} and the current best configuration. If the candidate configuration is better than the current configuration, it implies that there is a high possibility that we can find a better configuration from the neighborhood with center configuration C_{candi}. Otherwise, the algorithm will shrink the neighborhood size with the same center point C_{cur} with shrink factor f. The local search phase is terminated until a global search finds a local point with a neighborhood size smaller than a predefined

threshold N_t. This implies that the algorithm finds a local optimal point.

After the local search phase, the algorithm enters the global search phase again to find a promising area. If it finds a point that is better than the current local optimal configuration, the system enters the local search phase to refine the search, otherwise after a specified number of iterations g, the algorithm terminates.

There are several challenges we have to address in order to incorporate this gray-box hill climbing algorithm into our online tuning system. In the following, we discuss these challenges and our proposed solutions.

Mapping sampled configurations to tasks: Our monitor keeps track of the launched tasks and their associated configuration, as well as queued tasks both with and without assigned configurations. Given the fact that the tasks are independent from each other in YARN, when the configurations are generated, our tuning system randomly chooses a task from the queued task list and assign one of the configurations to the task. The configuration is then further adjusted based on the task related information.

Estimating cost of executed tasks: Equation 4.1 shows how we estimate the cost of each task. We consider four factors: CPU utilization, memory utilization, ratio of the number of spill records to the number of map output or combiner output, and ratio of task execution time to the max task execution time. The goal of this formula is to reduce the task execution time and the number of spill records of all the tasks, while keeping the memory and CPU fully utilized. The allocated resources should be fully utilized while allocating more resources can compete with other tasks thus lowering down resource utilization.

$$y = (1.0 - u_{mem}) + (1.0 - u_{cpu}) + num_{spill}/num_{mapoutput} + T_{task}/T_{maxtask}. \tag{4.1}$$

Utilizing tuning rules to reduce the number of convergence iterations and resizing the neighborhood size: We consider the statistics collected from the monitor for enhancing search quality. We detail the tuning rules in Section **??**.

Moreover, the dependencies between the parameters are also considered into the algorithm. For example, the memory size of map should be always greater than the memory size of *io.sort.mb*. The *mapreduce.reduce.shuffle.input.buffer.percent* should be always greater than the *mapreduce.reduce.shuffle.merge.percent*.

Since a job with a small number of map tasks can restrict mrOnline to try out all the parameters listed in table 4.2, considering these tuning rules helps us to converge to a suitable

configuration quickly. In the evaluation section, we quantify how the tuning effectiveness would be affected by the length of the job.

Configuration Parameters	Default Value
Memory tuning	
mapreduce.map.memory.mb	1 GB
mapreduce.reduce.memory.mb	1 GB
mapreduce.task.io.sort.mb	100
mapreduce.map.sort.spill.percent	0.8
mapreduce.reduce.shuffle.input.buffer.percent	0.7
mapreduce.reduce.shuffle.merge.percent	0.66
mapreduce.reduce.shuffle.memory.limit.percent	0.25
mapreduce.reduce.merge.inmem.threshold	1000
mapreduce.reduce.input.buffer.percent	0.0
CPU tuning	
mapreduce.map.cpu.vcores	1
mapreduce.reduce.cpu.vcores	1
mapreduce.task.io.sort.factor	10
mapreduce.reduce.shuffle.parallelcopies	5

Table 4.2 The key configuration parameters tuned in mrOnline.

4.5 Tuning Rules

Section 4.4 discussed means for finding a desirable configuration using gray-box based hill climbing algorithm. In this section, we present the guidelines that we incorporate into the algorithm for tuning for our two target use cases. Here, we focus on CPU and memory related parameters, as shown in Table 4.2. Other parameters are tuned using the hill climbing algorithm without incorporating additional tuning rules.

The tuning rules are aimed at improving the cluster resource utilization by adjusting the container resource to meet the task requirement and alleviate over- or under- utilization, as well as to reduce extra I/O traffic by carefully tuning the memory buffer options. The current implementation of mrOnline provides per task configuration, with application wide auto-configuration, e.g., selection of the number of mappers and reduces, being the focus of our ongoing work.

4.5.1 Tuning Guideline for the Two Use Cases

Use Case One: The goal in this case is to reduce the number of test runs and find a near optimal configuration for the YARN application. mrOnline is temporarily allowed to have worse performance than the default configuration as it searches through the configuration space comprehensively and monitors the changes and their impact. Therefore, we adopt an aggressive strategy to try out as many cases as possible in a wave pattern. We first update several parameters at once, and then adjust the parameter setting strategies in the next wave based on the collected statistics from the previous wave. Moreover, mrOnline controls the YARN application execution flow by holding off the launching of new tasks until the tasks in the previous waive are finished. This strategy slows down the test run execution, but allows the gray-box search algorithm to find a near optimal configuration with high confidence.

Use Case Two: In this case, we target to improve job performance in a single run. Thus, we adopt a conservative approach. We start the job with default values in the first wave and tune the parameters based on the collected information in the next. Moreover, mrOnline does not interrupt the application task scheduling sequence, thus minimizing the negative impact of the gray-box algorithm. The slave configurator running on each slave node uses the updated configuration file if available. If the configuration file is not present, the task is launched with default configuration.

4.5.2 Memory Tuning

The first part of Table 4.2 shows parameters that decide the memory allocation of map and reduce tasks and the memory allocation for the sub-phases within those tasks. These parameters need to be chosen carefully. If the memory is set to too big, it would waste memory resources that can be allocated to other containers, thus reducing the cluster utilization. In contrast, if the memory is set to too small, it would incur resource contention leading to extra disk operations (even out of memory errors), thus degrading performance. The optimal values of these parameters depend on the input data size, the map/reduce function, and the output data size.

To tune the memory allocation of map and reduce tasks, we adjust *mapreduce.map.memory.mb* and *mapreduce.reduce.memory.mb*. For aggressive tuning, we obey the hill climbing algorithm using LHS sampling to try memory options within the predefined memory range. After we get the task execution time and the memory utilization of map or reduce tasks that ran in

the previous wave, we adjust memory bounds as follows to help our hill climbing algorithm to narrow down the search space of these two parameters. If we observe memory utilization to be beyond 90% that may cause over-utilization, we increase the memory lower bounds to the 80 percentile of sampled memory value. We also decrease the memory upper bounds to 80 percentile of sampled memory value when detecting memory under-utilization (50% of memory utilization). When the tasks have data skew and exhibit heterogeneous behavior, mrOnline keeps track of the 80 percentile value, and adjusts the bounds based on them. For conservative tuning, we try different memory values only when they have high probability to yield better results. For the first wave, we conservatively use the default value and collect statistics. We then estimate the memory size needed by the map or reduce tasks using this information. If memory is underutilized, our hill climb algorithm tries the lower value with a higher probability, otherwise, it tries the higher value with a higher probability.

The next finer grain level of memory parameter tuning includes three key parameters: *mapreduce.task.io.sort.mb* in map phase, and *mapReduce.reduce.shuffle.input.buffer.percent* and *mapReduce.reduce.input.buffer.percent* in reduce phase. These memory allocation parameters affect the application efficiency in that they control the number of spill records written to local disks. If enough memory is allocated both in map and reduce phases, the number of spill records would be minimized. The *mapreduce.tasks.io.sort.mb* should not exceed the memory size of map tasks.

Ideally, the number of spill records in map phase should equal the number of map output records. The number of spill records in reduce phase should equal zero. Otherwise, the number of spill records is 3× the number of map output records in the worst case. However, allocating more than needed memory would cause memory contention between the buffers and application logic, which negatively impact the MapReduce job performance. The optimal size of memory buffer depends on the job characteristics and cluster resource information.

The approach used for tuning *mapreduce.task.io.sort.mb* is to configure the buffer size based on map output size by continuously monitoring the number of spill records and the size of map outputs. For conservative tuning, the value is set as the default value in the beginning. As the first few map tasks are started, the buffer size is set to the estimated map output size. If the ratio of increased number of spill records to increased map output records is greater than one, we increase the lower bound to 80 percentile of the sampled values, since the current parameter value is not big enough to hold the map or combine outputs. If the ratio is one, mrOnline decreases the upper bound to 80 percentile of the sample values. For aggressive tuning, the rule is similar, except that before we get any statistics, we try multiple

values as determined by our hill climbing algorithm.

The parameter *mapreduce.map.sort.spill.percent* decides when to spill out mapreduce data to disk. It enables pipelining between map functions and disk writes. When the *io.sort.mb* is big enough, the value of *mapreduce.map.sort.spill.percent* should be set high to ensure that disk writes are not triggered. Thus, for both aggressive and conservative tuning, we set the value to 0.99. If spilling extra records is unavoidable, we reset the value to default.

For tuning buffers in reduces phase, we calculate the buffer size based on the estimated reduce input size. For conservative tuning, in the beginning, the value is set as default, the reduce input size is estimated by monitoring the number of spill records and the sum of size of partitions generated by each map output to a particular reducer.

The parameter $MapReduce.reduce.input.buffer.percent$ decides when to write the merged reduce output to disk. For example, when the reduce function requires only small amount of memory, $mapreduce.shuffle.input.buffer.percent$ is set equal to the shuffle buffer to avoid any spills written to disk. Specifically, we use the memory utilization statistics from node manager to decide the memory usage of reducers.

The parameter *mapreduce.reduce.shuffle.merge.percent* controls the trigger of memory to disk merge pipelining shuffle and memory-disk merge. It cannot exceed the reduce buffer size. For conservative tuning, the value is initially set as default value, when shuffle buffer is big enough to accommodate all the reduce input, the value can be set equal to shuffle buffer to avoid additional disk I/Os. Otherwise, for safety, the value is set to $mapreduce.reduce.shuffle.input.buffer.percent - 0.04$ which has the same value difference with *mapreduce.reduce.shuffle.input.buffer.percent* in YARN default configuration.

Finally, we set *mapreduce.reduce.merge.inmem.threshold* to 0, which allows Hadoop to depend only on memory consumption to trigger the merge.

4.5.3 CPU Tuning

YARN supports allocating different number of CPUs to map and reduce tasks. The parameter *yarn.nodemanager.resource.cpuvcores* manages the number of CPU virtual cores that can be allocated for containers running in each slave node. If the value is 32, then on a 8-core machine, each virtual core has 1/4 share of a physical core. Given that the number of

physical cores per machine is fixed, a larger value yields smaller share per virtual core. This parameter is not suitable for dynamic tuning.

The parameters *mapreduce.map.cpu.vcores* and *mapreduce.reduce.cpu.vcores* directly control the CPU allocation of map and reduce tasks. The basic tuning rule is to allocate enough CPU resources to map and reduce tasks without sacrificing the cluster utilization. For conservative tuning, we start with the default value of 1, and collect container utilization information from the node manager. If full CPU utilization is observed, we increase the allocation by 1. If the task execution time is reduced and cluster-wide CPU utilization does not decrease, we continue to increase the virtual core allocation.

The parameter $mapreduce.reduce.shuffle.parallelcopies$ determines the concurrent transfers executed by reduce tasks during shuffle. The desirable value depends on the amount of shuffled data. Higher amount leads to high number of parallel shuffles. For conservative tuning, starting from default value, we increase the parameter in increments of 10 until the task execution time is not improved any further.

The parameter $mapreduce.task.io.sort.factor$ controls the concurrency of disk to disk merge with a default value of 10. The optimal value depends on the amount of data to be merged. For conservative tuning, we increase the value by 20 until the task execution time stops showing improvement.

This summarizes all the guidelines we incorporate into mrOnline for parameter tuning. The provided API of the dynamic configurator is flexible, and can be used to easily incorporate additional tuning logic for more parameters as necessary.

4.6 Implementation

We describe the implementation of mrOnline in this section. We implement mrOnline based on Yarn-2.1.0. The online tuner is implemented as a daemon which includes the three components mentioned in Section 4.2 as three daemon threads. More specifically, the three components are implemented by extending the AbstractService class within YARN. The AbstractService is a class that maintains a couple of service state and a list of service state change listeners. Once the service state has been changed, the service state change listeners are informed. The online tuner is implemented by extending CompositeService class within YARN. The CompositeService is a class that consists a list of AbstracService. It has a

shutdown hook that allows the child services within the composite service to be shut down gracefully when the JVM is shut down. Leveraging this feature, the online tuner and its child components can be shut down gracefully.

The monitor periodically gets each job counter from YARN through JobClient interface. It then sends the job ID and job counters to the tuner. The monitor also gets the task level counter and cluster level information such as the CPU, memory, network I/O, disk I/O from each slave nodes.

The tuner takes the input from the monitor and decides what parameters have to be changed and what values should these parameters be set to by using gray-box hill climbing algorithm and tuning rules described in Section 4.4,4.5.

After the tuner generates the list of parameters needed to be changed, it sends them to the dynamic configurator. The dynamic configurator sends the list of parameters to YARN through the JobClient interface. The hacks implemented in YARN then pick up the values and change the parameters accordingly.

4.7 Evaluation

In this section, we show the effectiveness of our approach on a 19 node cluster. We start by demonstrating the performance improvement of MapReduce applications using aggressive tuning strategy of mrOnline for use case one. We then show that mrOnline can generate desirable configurations yielding better application performance using conservative performance tuning for use case two. Next, an experiment illustrating the impact of job size on the effectiveness of mrOnline is conducted. Finally, we exhibit that mrOnline can also improve application performance in a multi-tenant environment.

4.7.1 Methodology

Each node on our 19 node cluster has two Intel Quad-core Xeon E5462 2.80 GHz CPU, 12 MB L2 cache, 8 GB memory, a 320 GB Seagate ST3320820AS_P SATA disk, and a gigabit network card. One node works as the master and the rest 18 nodes work as slaves. Nine nodes are on one rack while ten nodes are on the other rack. Two racks are connected with a 24 port switch.

Benchmark	Input Data	Input Size	Shuffle Size	Output Size	#Map, #Reduce	Job Type
bigram	wikipedia	90.5G	80.8G	27.6G	676,200	Shuffle
inverted-index	wikipedia	90.5G	38G	10.3G	676,200	Map
wordcount	wikipedia	90.5G	30.3G	8.6G	676,200	Map
text search	wikipedia	90.5G	2.3G	469m	676,200	Compute
bigram	freebase	100.8G	84.8G	77.8G	752,200	Shuffle
inverted-index	freebase	100.8G	21G	11G	752,200	Compute
wordcount	freebase	100G	16.7G	9.4G	752,200	Map
text search	freebase	100.8G	906m	229m	752,200	Compute
Terasort	synthetic	100G	100G	100G	752,200	Shuffle
bbp	N/A	0	252K	0	100,1	Compute

Table 4.3 The benchmarks and their characteristics.

We compare against the default configuration in terms of 1)job execution time; 2) spill records; 3) CPU utilization and I/O utilization. The parameters used are mostly the same as the values specified at Hadoop Wiki [19]. We use block size of 128 MB, number of virtual cores available for container allocation is 28 (4 for data node and node manager daemons), memory available for container allocation is 6 GB (2 GB for data node and node manager daemons).

Table 4.3 shows the MapReduce applications we used to evaluate mrOnline. In addition to Terasort, word-count(WC), text search (Grep) and BBP which is distributed with Hadoop release, we add two more interesting applications bigram and inverted index. Bigram [103] counts all unique sets of two consecutive words in a set of documents. Inverted-index [103] generates word to document indexing from a list of documents. BBP is a map/reduce program that uses Bailey-Borwein-Plouffe to compute exact digits of Pi, it is a compute intensive application. We classify the applications into three categories which include Map intensive, Shuffle intensive and Compute intensive. Map intensive means that map phase consume the most large part of the execution time mostly doing I/Os. Shuffle intensive jobs spend the largest part of time in shuffle phase while compute intensive jobs spendslargest part of time in Map phase doing computation.

We use two data sets for the four applications including bigram, inverted-index, wordcount and text search. Wikipedia [104] have the original data set size of 45 GB. We concatenate two copy of Wikipedia data set together to generate a larger data set size of 90 GB. Note that this will not change the workload characteristic of the dataset. Freebase [105] is an open source data set released by Google. It is a knowledge graph database for structuring human

knowledge which is used to support the collaborative web based data oriented applications. The dataset used by Terasort is generated by Teragen.

To consider the variance due to irregular events such as network, disk I/O congestions, hardware and file system errors, we repeat each experiment for four times, and use the average value.

4.7.2 Performance Improvement of Use Case One

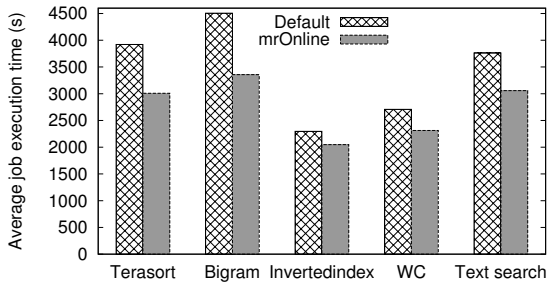

Figure 4.3 Comparison of job execution time for mrOnline and default configuration on Wikipedia data set for Use Case One.

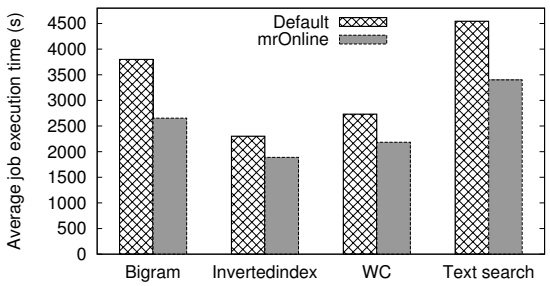

Figure 4.4 Comparison of job execution time for mrOnline and default configuration on Freebase data set for Use Case One.

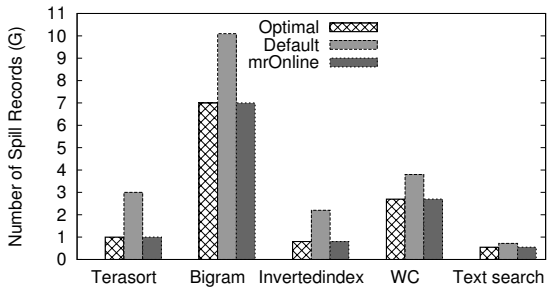

Figure 4.5 Comparison of number of spill records for mrOnline and default configuration on Wikipedia dataset.

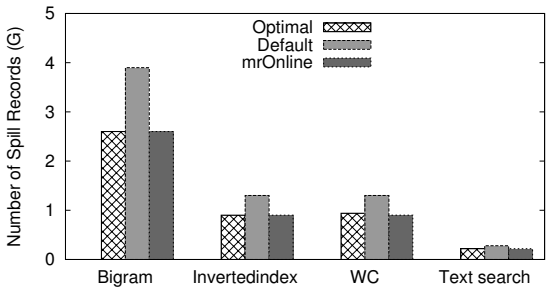

Figure 4.6 Comparison of number of spill records for mrOnline and default configuration on Freebase dataset.

In this experiment, we evaluate the effectiveness of mrOnline for use case one which use aggressive tuning. We first run mrOnline together with each application, and get a best parameter configuration generated by mrOnline. We then use the configuration to run the application and compare against applications running with the default configuration. Since

we have around 600−800 number of maps and 200 number of reducers, we finish our gray-box based hill climbing algorithm within a test run.

Figure 4.3,4.4 show the average job execution time of the four applications on data set wikipedia and freebase besides application Terasort in Figure 4.3. The job execution time is improved by 23%,25%,11%,14% and 19% for Terasort, bigram, inverted index, word-count, textsearch respectively. For freebase dataset, the performance enhancement is 30%, 18%,20%,25% respectively. mrOnline improves the performance mainly due to three factors: 1) it effectively reduces the number of spill records written and read from disks; 2) it improves the resource utilization by tuning the container size for mappers and reducers; 3) it detects near-optimal value for other performance related parameters.

To further understand the effectiveness of mrOnline, we present how mrOnline reduce the number of spill records. Figure 4.5,4.6 show the number of spill records generated by Map tasks using configuration of mrOnline comparing against default configuration. X-axis shows the four applications on wikipedia and freebase dataset while Terasort in Figure 4.5 uses synthetic dataset. Y-axis shows the number of spill records in Giga. *Optimal* refers to the number of records generated by combiner in Map phase or generated by Map function if combiner is not existed. It represents the optimal number of spill records an optimal configuration would produce. We can see that the spill records are effectively reduce to optimal results for all applications on both Wikipedia and Freebase dataset and for Terasort.

4.7.3 Performance Improvement for Use Case Two

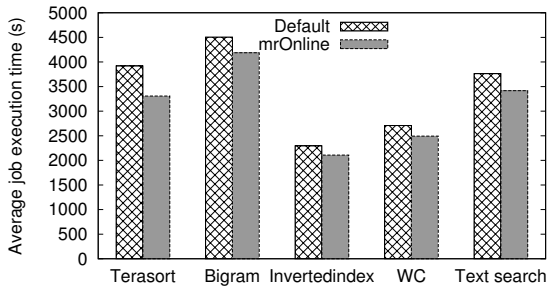

Figure 4.7 Comparison of job execution time for mrOnline and default configuration on Wikipedia dataset for Use Case Two.

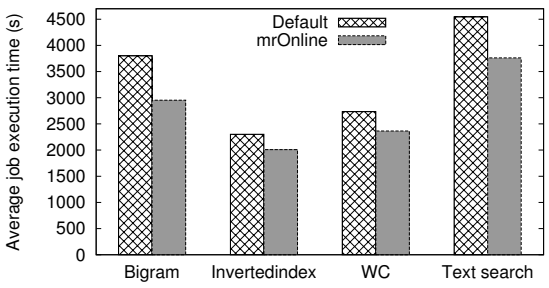

Figure 4.8 Comparison of job execution time for mrOnline and default configuration on Freebase dataset for Use Case Two.

In this experiment, we compare the job execution time of mrOnline using the conservative

tuning strategy comparing against YARN default configuration on Wikipedia, Freebase and synthetic data sets. Conservative tuning is beneficial for applications that run once since the goal is to improve performance but not to find the best configurations. We run mrOnline together with the applications and measure the job execution time. For synthetic data set of Terasort and Wikipedia data set of other four applications, mrOnline reduces the execution time by 16%, 7%, 8%, 8%, 9% respectively. For data set Freebase, mrOnline shows a similar trend which improves performance by 22%, 13%, 14%, 17% respectively. mrOnline reduces the execution time by up to 22% because it improves the cluster utilization by adjusting the container size, alleviates the I/O contention by reducing the spill records and searches for the optimal values for other performance related parameters.

This experiment demonstrates that mrOnline can effectively reduce job execution time for applications that run once or few times. Users do not need to worry about tuning application parameters before running jobs enjoying a free ride on performance speedup.

4.7.4 The Impact of Job Size on Performance Tuning Effectiveness

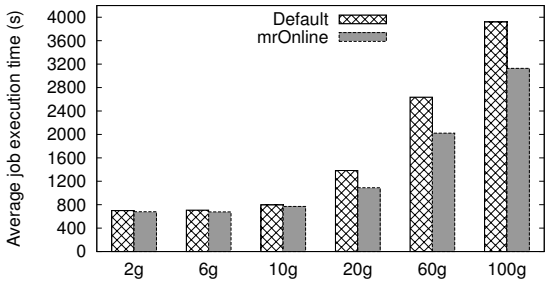

Figure 4.9 Comparison of job execution time for mrOnline and default configuration on Terasort with different data set size .

We next study how the tuning effectiveness of mrOnline is impacted by job size. In this experiment, we use Terasort and run it with different sizes of input data sets ranging from 2 GB to 100 GB. The number of reducers is around 1/4 of the number of mappers. For example, we have 4 reducers and 16 mappers for a job with a size of 2 G, 12 reducers and 46 mappers for another job with a size of 6 G. We run mrOnline for a single run together with each job and generate a configuration using aggressive tuning. We then use this configuration to run the job again and compare against YARN default configuration. We can see

that mrOnline reduces job execution time marginally for jobs with sizes smaller than 10 GB. The reason is that mrOnline does not have enough number of mappers or enough number of reducers to search through the configuration space. Jobs finishing before mrOnline fail to find good configurations. For jobs that are greater than 20 G, mrOnline becomes effective and reduces job execution by 21%, 23%, and 20% for Terasort with job sizes of 20 G, 60 G and 100 G respectively. After the job input data set size is greater than 20 G the effectiveness of mrOnline does not further improve since the numbers of mappers and reducers are sufficient for mrOnline to find near-optimal configurations.

4.7.5 The Tuning Efficiency for Multi-tenant Environment

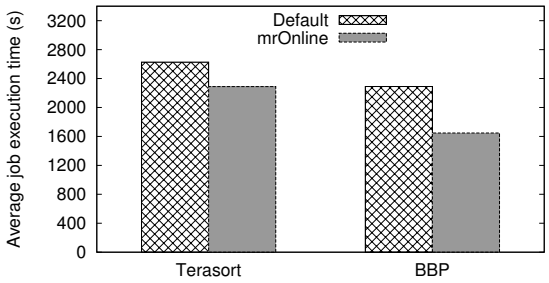

Figure 4.10 Job execution time of Terasort and BBP.

In our next experiment, we demonstrate that mrOnline is particularly useful in a multi-tenant environment. We run two MapReduce applications Terasort and BBP simultaneously using fair sharing scheduling algorithm. We configure Terasort with an input data set size of 60 GB using 448 mappers and 200 reducers and configure BBP to compute 0.5 M digits. We then execute mrOnline with aggressive tuning and produce desirable configurations for the two applications. Figure 4.10, 4.11, 4.12 represent the experimental results collected by running applications using configurations generated from mrOnline and using YARN default configuration.

Figure 4.10 shows the job execution time of Terasort and BBP. We can see that mrOnline reduce job execution time by 13% and 28% for Terasort and BBP respectively.

To further understand the performance impact of mrOnline, we examine the memory utilization and CPU utilization of Terasort and BBP. Figure 4.11 illustrates the memory utilization while Figure 4.12 shows the CPU utilization. Terasort-m stands for the average utilization

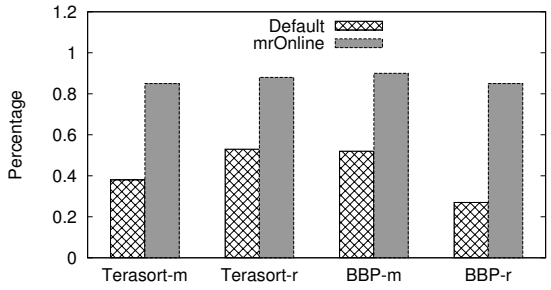

Figure 4.11 Memory utilization of Terasort and BBP.

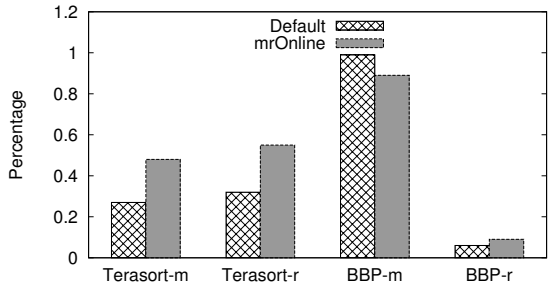

Figure 4.12 CPU utilization of Terasort and BBP.

of all Mappers while Terasort-r stands for the average utilization of all Reducers. BBP-m and BBP-r have a similar meaning yet they are for application BBP. We observe that, by using the default configuration, the memory utilization of Terasort and BBP is below 50%. In contrast, mrOnline improves the memory utilization of the two applications to above 80% for both map tasks and reduce tasks. For CPU utilization, we see that, except the mappers of BBP, all the CPU utilization is below 25%. mrOnline improves the CPU utilization by assigning fewer CPUs to Terasort and reducers of BBP. Note that the CPU utilization of the mappers of BBP is around 99%. mrOnline identifies this as CPU over-utilization, and allocates more CPU cores to BBP. Moreover, we note that the number of spill records of Terasort is reduced from 1.8 G to 0.6 G as compared to the default configuration. Reducing the number of spill records is beneficial especially when disk I/O is the performance bottleneck.

This experiment shows that mrOnline can effectively increase the memory utilization and CPU utilization for Terasort and BBP and thus reduce the job execution time. In other words, in this multi-tenant experiment where CPU is a bottleneck for BBP, mrOnline successfully identifies idle CPUs and requests part of them to BBP. Thus, we have demonstrated that mrOnline can mitigate hot-spots in the cluster and improve system utilization.

4.8 Chapter Summary

While MapReduce job parameter configuration impacts performance significantly, in current implementations, the parameter tuning burden is placed on the application programmers. This is not ideal, especially because the application programmer may not have enough system-level expertise to select the best configuration, consequently leading to system inefficiency and degraded application performance. In this paper, we presented the design of

mrOnline, a platform that enables task-level dynamic configuration tuning to improve performance of MapReduce applications. mrOnline expedites the test runs by trying out multiple configurations within a single test run. Given the large MapReduce parameter space, how to effectively find a near optimal solution is challenging. To this end, we proposed a gray-box based hill climbing algorithm to systematically search through the space and find a desirable configuration. To speedup the convergence iteration of our hill climbing algorithm, we leverage MapReduce runtime statistics and design tuning rules for some of the key parameters. We have implemented mrOnline on the YARN framework, and our evaluation shows that on a 19-node cluster and across a suit of five representative applications, mrOnline achieves an average performance improvement of up to 30%.

Chapter 5

Cooperative Storage-Level De-Duplication for I/O Reduction

We have introduced resource management techniques we designed for hosting MapReduce clusters in the cloud in Chapter 3 and Chapter 4. We then discuss the approaches we devised to improve the storage scalability and efficiency for VDEs in this and next chapters.

Data centers are increasingly being re-designed for workload consolidation in order to reap the benefits of better resource utilization, power savings, and physical space savings. Among the forces driving savings are server and storage virtualization technologies. As more consolidated workloads are concentrated on physical machines e.g., the virtual density is already very high in virtual desktop environments, and will be driven to unprecedented levels with the fast growing high-core counts of physical servers the shared storage layer must respond with virtualization innovations of its own such as de-duplication and thin provisioning. A key insight is that there is a greater synergy between the two layers of storage and server virtualization to exploit block sharing information than was previously thought possible.

In this Chapter, We reveal the synergy via developing a systematic caching system, Sea-Cache, to explore the storage and virtualization servers interactions. We also quantitatively evaluate the I/O bandwidth and latency reduction that is possible between virtual machine hosts and storage servers using real-world trace driven simulation. Moreover, we present a proof of concept NFS implementation that incorporates our techniques to quantify their I/O latency benefits.

5.1 System Design

This section describes the key design aspects of SeaCache and how de-duplication information is cooperatively shared between storage server and hosts.

5.1.1 Design Rationale

A traditional approach to I/O bandwidth saving is to not modify the host and storage server software stack, instead to introduce dedicated nodes for de-duplication, i.e., de-dup boxes [82], both at the host and the storage server. These boxes keep track of the data blocks through a content sharing information/index (CSI) database of all the blocks they have been sent and received, and work together to avoid writing multiple copies of data to the disk. A CSI entry usually consists of a block identifier and the corresponding hash value calculated using collision resistant hash functions such as SHA-1 [106]. Similarly, SeaCache assumes that hash collision from SHA-1 is lower than memory bit flip errors due to cosmic rays for all practical purposes.

In the de-dup box approach, the box is a separate entity and it is not aware of the host-side or storage server-side cache contents. Thus, read requests from different clients will always be sent to the storage server even if the data already exists in the host cache. Furthermore, the storage server side block de-duplication information is not leveraged, and thus, this data is maintained separately at the de-dup boxes and at the storage server. By integrating host-side cache with server-side de-duplication, we can explore the opportunity to build a cooperative I/O de-duplication solution between host and storage server.

5.1.2 Architecture

Figure 5.1 shows the overall architecture of SeaCache. The target environment comprises host physical machines with multiple clients (VMs), which interact with a storage server for persistent data storage. The main software components include a specialized *page cache manager* on the host, a *de-duplication system* on the storage server, a storage server *cache-tracker* that keeps track of the host cache contents, and a *protocol for sharing CSI* between the hosts (cooperative caching) and the shared storage server.

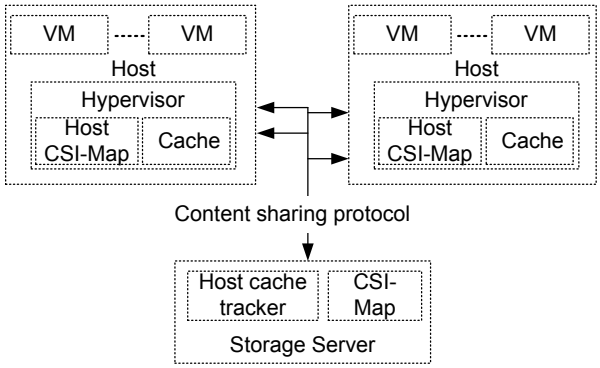

Figure 5.1 SeaCache system architecture overview.

When data is written to the storage server, it is de-duplicated (either in-line or as a background process) as follows. The contents are hashed at the granularity of a block, and the hash information is saved in the CSI data structure. The CSI is then compared to and, if not already present, stored in a CSI database. Each entry of the CSI database is a tuple consisting of the logical block number (LBN), and the block's hash value. If a logical block's CSI matches one already in the CSI database, it indicates that the logical block is a duplicate and its contents are not written to the disk. Thus, a physical block could potentially map to multiple logical blocks. The information about mapping of logical blocks to a physical block is maintained by the storage server in a mapping structure, *CSI-Map*, which has an entry for every physical block (in use).

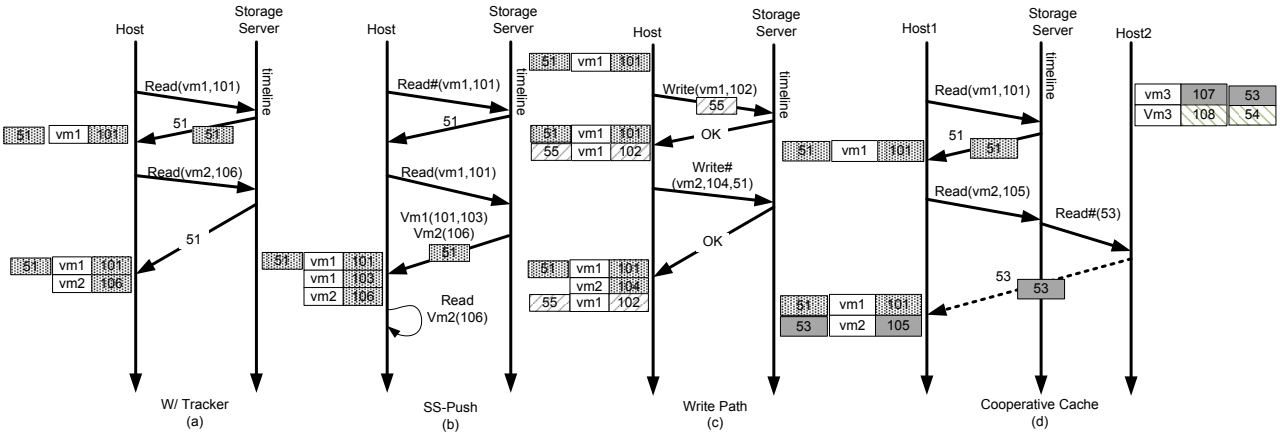

Figure 5.2 SeaCache protocols using CSI-Map information.

Figure 5.2 illustrates the read and write protocols included in SeaCache. We detail each protocol in the next few sections.

5.1.3 Read Protocols in SeaCache

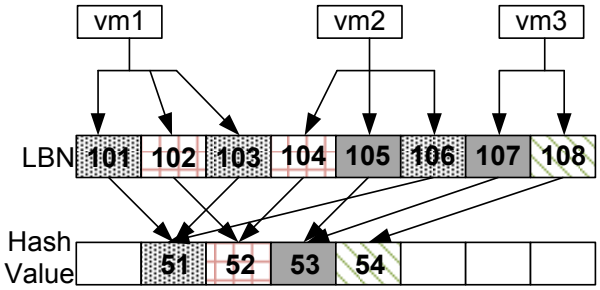

Figure 5.3 An example logical to physical block mapping.

In the following discussion, we present our read protocols using an example physical block usage scenario illustrated in Figure 5.3.

Basic Protocol When a logical block, e.g., 101, for VM_1 is not found in the host's cache, the host first requests the CSI for the LBN (`Read#`) from the storage server, instead of sending a regular read request. The storage server looks up the associated CSI-Map entry and returns the content identifier, which in this example is Hash 51. The host maintains a local CSI-Map cache, which it uses to determine that Hash 51 is not present at the host. The host then sends an actual data request for LBN 101. Upon receipt of the block from the storage server, the host also updates its local CSI-Map cache to store the mapping information. Later, when the host sends a CSI request for LBN 106 for VM_2, the response Hash 51 is already in the cache, and a subsequent read request to storage server for LBN 106 is avoided.

This basic protocol saves network bandwidth between the host and the storage server by avoiding the necessity to put the data block on the wire. However, it introduces an extraneous round of CSI request messages to be exchanged for every read I/O request missed from client caches.

Protocol with Tracker We can eliminate CSI requests by keeping track of the content of the host cache using the cache-tracker at the storage server. Figure 5.2(a) displays the storage server-Host protocol using this approach. Here, the host uses a traditional request for reading LBN 101 of VM_1. The storage server replies with data and its hash, Hash 51, back to the host. The storage server learns that Hash 51 is stored in host cache. Later, when the host requests a read for LBN 106 for VM_2, the storage server simply returns the CSI-Map entry that points to the already present Hash 51 in the host cache.

In this approach, we require the storage server to accurately track the host cache contents

by the read requests and eviction information. Therefore, whenever the host evicts data from its cache, it needs to inform the storage server by piggybacking eviction information on its next message to the storage server (not shown in the figure). Note that no additional separate message is needed. This approach facilitates the cooperative cache between hosts (detailed in Section 5.1.5). This optimization avoids additional round trip time (RTTs) for CSI requests for which the host has to communicate with the storage server to get the hash value of the requested block.

Maintaining a CSI-Map within the storage server might seem to consume additional memory. However, such tracking is worthwhile because it allows the storage server to leverage the hosts cache space to exploit more de-duplication and thus reduce read I/O. In our design, each data block is 4 kilobytes, while hash-entry is 24 bytes. Therefore, we can map 170× more blocks in cache using the CSI-Map instead of caching the actual data. Handling the additional eviction information for remote cache-tracker of hosts may require additional storage server CPU cycles. However, we argue that there is a large disparity in computational power versus I/O latency. For instance, a 3 GHz processor has 3 million compute cycles to spare for every 1 millisecond of latency from the I/O subsystem. Therefore, we argue that compute overhead is not an issue in this case.

Storage Server Push request, *storage server-Push*, or via an asynchronous callback that sends the associated CSI-Map entry from the storage server to the host. Figure 5.2(b) illustrates the storage server-Push. In this case, when the host requests the data for LBN 101 for VM_1, the storage server replies with not only the data but also the CSI-Map entry for the associated hash value, which in this case indicates that LBNs 101 and 103 of VM_1 and LBN 106 of VM_2 all map to Hash 51. The extra information is stored in the host's CSI-Map cache. Later, when the host wants to read LBN 106 for VM_2, it already knows that the associated hash value Hash 51 and that its contents are already in the host cache. Thus, no extra CSI request is sent. The CSI-Maps piggybacked by the storage server determines the quality of I/O reduction using this approach. If the storage server is aware of the topology information of the data-center/cloud, the storage server can choose to send back the CSI entry of VMs on a particular host, improving the I/O reduction. Furthermore, it is better to send back the CSI information that resides in the cache only. Thus, this approach is sensitive to the knowledge and understanding of the workload characteristics to obtain optimal performance.

5.1.4 Write Protocol in SeaCache

Consistency In virtual environments, there are no shared writes. Each virtual machine will only be able to access the virtual image file attached to it. Therefore, we focus our discussion to such share-nothing environment.

Figure 5.2(c) demonstrates a basic protocol for I/O reduction on the write path, which is similar to the read path protocol. Initially the VM_1 on the host writes a block with LBN 102 whose original hash value is Hash 52. The host calculates the new hash value Hash 55, determines that it is not in its local cache and sends a request to write the data to the storage server. Now, the second write request by VM_2, LBN 104, with hash value changed from Hash 52 to Hash 51 is a cache hit. In this case, the host sends only the metadata to the storage. If the storage server is able to map the hash value to the actual data, it replies with success and no further action needs to be done. If the storage server replies with failure, the host will now need to send the actual data.

In order to check whether a block represented by the hash value sent back by any host is actually present, the storage server needs to perform a CSI-Map lookup. Most storage servers cannot keep the entire CSI-Map of their blocks in the primary caches. Any design or solution to seek CSI-Map from the secondary-level cache will add additional latency to the I/O request. If the storage server performs I/O reduction on only CSI maps in the primary cache, some write path I/O reduction opportunities will be missed.

By integrating content sharing information into storage server and host, the de-duplication workload can be distributed across hypervisor and storage server. Once the hash value of the blocks are calculated the storage server can simply leverage that information to reduce the usage of computational and disk I/O resources, which in turn can benefit the foreground read/write requests service.

5.1.5 Protocol for Cooperative Cache in SeaCache

It is straight forward to expand the read/write CSI protocol to build a cooperative cache between multiple hosts and storage server based on our deployment architecture as described in Section 5.1.2. Cooperative cache aggregates the cache space from all hosts to further distribute the I/O load away from the storage server. To implement such cooperative our deployment architecture as described in Section 5.1.2. Cooperative cache aggregates the

cache space from all hosts to further distribute the I/O load away from the storage server. To implement such cooperative cache, efficient meta-data lookup and request forwarding mechanisms are needed. The host cache tracker offers an ideal data structure for meta-data lookup. If more than one host is keeping the required data, the storage server can randomly choose one to forward the requests to.

Figure 5.2 (d) describes the working of SeaCache. When the first read request from host 1 arrives at the storage server, the storage server checks for the block(Hash 51) in its cache. If the block is not found, the server checks its remote cache-tracker for availability of the block in any of the remote hosts. In the above figure, the hash value Hash 51 is not found in the cooperative cache. Therefore, the storage server fetches the data block from disk and sends back both Hash 51 and the data block. When a second request arrives, the storage server determines that the hash value of the requested block is Hash 53, which is not stored in its cache but stored in the cache of host 2. The server delegates the block request to the client at host 2, which then sends the data directly to host 1.

SeaCache requires host 1 to directly receive data from host 2, which is not supported by traditional RPC calls. However, RPC delegation as proposed by Anderson et. al. [107] should suffice as an elegant alternative here. The delegation protocol creates a reply token, allowing the reply token to be relayed from node to node until some node answers the request. Without this technique, we would need two RPC calls one to communicate with the storage sever and another one to communicate with host 2 to get the data.

We can further mitigate the cache-misses in SeaCache protocol by **enhancing the cache replacement algorithm** inside the host. The enhanced LRU or ELRU host cache manager uses CSI information from storage server to determine the block to evict from its cache. Instead of evicting the least recently accessed block, this algorithm also weighs in the sharing count of the block before evicting it. Specifically, we choose the least shared block within the last n blocks to be evicted. A larger n may yield a higher hit ratio but may have a higher eviction overhead. In Section 5.3, we explore the impact of n on the efficiency of the host cache algorithm. This sharing count metric becomes important in context of SeaCache as it is more likely to improve the cache hit for highly referenced blocks.

5.2 Experimentation Methodology

In this section, we discuss both a simulator and a proof-of-concept implementation of SeaCache, and the workloads we have used for experimentation.

5.2.1 Simulator Framework

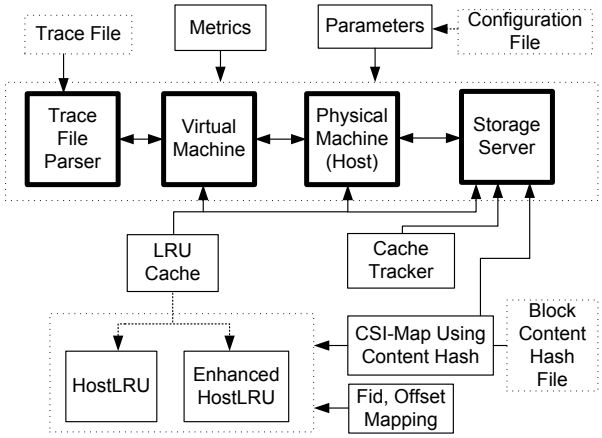

Figure 5.4 Simulation framework to evaluate SeaCache.

In order to test our protocols in a controlled setting and explore the large configuration space, we have built a realistic system-level simulator. Figure 5.4 shows the modules of the simulator and their interactions.

Trace File Parser: This main module takes preprocessed trace files as input, parses them, and reconstructs the read and write commands to drive the simulation.

Virtual Machine: This main module implements a model of the client **Physical Machine:** This is another main module that models a host using specified configuration settings. It supports a host cache that can be configured to use either a *HostLRU* or *enhanced HostLRU* caching policy.

Storage Server: This main module models a storage server with an LRU cache, host-cache content tracker, and CSI-Map sharing features. It further uses the **Cache Tracker** module to keep track of content of host caches.

Support Modules: These modules facilitate the main modules. The **LRU Cache** module

provides a content-based cache implementation that can be instantiated by the main modules as needed. The **Metrics** and **Parameters** modules are linked with all the main modules to enable flexible configuration, and produce different observable metrics. Specifically, the **Metrics** module keeps track of hit ratio, I/O bandwidth usage, latency of each request, total number of commands, number of read/write commands, number of logical blocks etc. The **Parameters** module takes charge of parsing the configuration file and setting up the corresponding module. Example parameters include cache size and policy, number of hosts, and number of storage server.

Finally, the configuration file provides an easy means for exploring the design space without modifying the source code. The modularity and flexibility of this framework greatly speeds up the simulation process.

5.2.2 Implementation

We have implemented a proof-of-concept read protocol prototype of SeaCache, specifically the CSI-Map sharing solution by modifying the NFS v3 protocol, client and server components in Linux 2.6.32.15, using about 1200 lines of *C* code.

The implementation setup comprises of Linux-based hosts running Oracle® Virtual Box 3.2.8 to provide client VMs. The clients run Windows XP SP3 with disks mounted via NFS. We use a write-through cache policy to ensure that we can use NFS v3 close-to-open consistency model. For computing CSI, we simply use the offset of the block and assume it to be a sufficiently unique content identifier. This is in-line with similar assumptions made in hypervisor design, which uses this concept to maintain a common base disk for multiple VMs by separating overwrites using snapshots for their VDI environments [108].

NFS Client: We trap *nfs_readpage(s)*, *nfs_readpage_result*, and *nfs_wb_page* NFS calls to enable CSI sharing and to service client block requests. The CSI-Map maintains two data-structures: (a) A *Fid-Offset* hashtable, which maps (file-handle, offset) to the actual PBNs; and (b) a *CSI hashtable*, which maps to a list of *Fid-Offset* entries that have the same content. The macro implementation of `uthash` [109] was used for the hashtable implementation. CSI-Map also supports an LRU list for removing (writing to disk) least-used entries if needed. The client *nfs3_xdr_readargs* and *nfs3_xdr_readres* RPCs are modified to marshal the SS-Push. Note that some of the data structures in CSI-Map have been built in anticipation of incorporating and integrating protocol information exchange with an NFS server

that supports a de-duplicated file system, such as OpenDeDup [110].

NFSD Server: The NFS server maintains an exception list for all files opened by a particular NFS client. This list identifies block offsets that have been modified by any of the open files. This information is marshalled into a RPC to the client by modifying the *nfss3svc_decode_readargs* and *nfs3svc_encode_readres*.

We instrumented the Linux kernel to identify the cache hits and misses to our cache as well as the latency observed by each request. For testing, the clients ran typical OS operations such as booting, virus scan, and compilation of source code.

5.2.3 Workloads

In this section, we briefly describe the real-world traces that we have used to drive our evaluation of SeaCache.

CIFS Network Traces includes I/O traces collected over a period of four months from two large-scale enterprise storage servers deployed at a company which uses Common Internet File System (CIFS) as the network protocol and hosts about 1500 employees.

VDI Traces comprise of traces collected for two weeks from a system that was supporting 9 VMs in an in-house Virtual Desktop Environment. Here, in order to separate user-generated I/O from other accesses we disabled any anti-virus program on all the VMs. The VMs read 17.3 GB and write 6.1 GB data per day on average.

Due to resource consolidation efforts, in addition to exhibiting general usage characteristics, the VDI environment exhibits some interesting spikes in I/O requirements at certain times of the day. In VDE, login/boot storms are generated on storage boxes where a large spike of read requests are created as users login/boot to their desktop. Such a storm is highly predictable because most corporate users start using their computers around the same time, e.g., 9 am. In certain VDEs, the scheduled 3 am virus scan triggers a virus-scan storm. These are read intensive storms that could be mitigated as most of the clients request identical set of blocks from the storage server. Similarly, write intensive storms such as a patch update or virus update can be configured to occur in the trace periodically.

Test-Dev Trace: An enterprise level test-development environment uses resources continuously to 1) deploy/compile code, 2) install builds, and 3) perform QA activities. The QA

integrated test environment often consists of a number of sub-environments where each sub-environment is associated with the testing of a particular feature being developed. Thus, at any given point in time, one of the above three processes are running for each of the QA sub-environments. We recreated such a test-dev environment for trace collection. In our setup, there were around 20 build-test VMs (each of which contained a sub-environment) on a physical host. Since most test-dev cycles are almost identical, we emulated a larger setup by replicating the traces and using simple Poisson arrival process to vary the start time of each instance of a test-dev cycle, finally giving us our Test-Dev trace. Within a single Test-Dev environment, while 40 MB data is read, 250 MB data is written. This is a write intensive trace.

5.3 Evaluation

We evaluate SeaCache using our simulator and our prototype implementation of Section 5.2.2. In our simulator, we configure the cache size of each VM to be 256 MB, each physical host to accommodate 10 VMs at most, and the cache size of storage server to be 4 GB. Most of our experiments measure the I/O bandwidth consumption between storage server and physical hosts, which are illustrated as **bars**, and the average latency of the I/O requests, which are illustrated as **stars** on the same graphs. In the following, we present the details of our experiments and observations.

5.3.1 CSI Sharing Protocol Analysis

In our first set of experiments, we use our simulator to analyze the different CSI sharing protocol algorithms described in Section 5.1 using the CIFS, VDI and Test-Dev workloads. The protocols being analyzed are as follows: 1) *Baseline* protocol that does not transfer any CSI information, and we use it as the baseline to compare to when reporting performance improvement results; 2) *Dedup-Read* approach used by de-duplication box on read path; 3) *W/Tracker* CSI sharing protocol algorithms described in Section 5.1 using the CIFS, VDI and Test-Dev workloads. The protocols being analyzed are as follows: 1) *Baseline* protocol that does not transfer any CSI information, and we use it as the baseline to compare to when reporting performance improvement results; 2) *Dedup-Read* approach used by de-duplication

box on read path; 3) *W/Tracker* where the storage server exactly tracks the host cache contents; 4) *SS-Push* where the storage server exactly tracks the host cache and pushes back more CSI to clients; 5) *Coop-Cache* using cooperative cache with SS-push; 6) *Dedup-RW* approach used by de-duplication box both on read and write path; 7) *SeaCache* in which all the proposed features are enabled including read, write path and cooperative cache. For these experiments, the file system block size was set to $4KB$, and the network and disk latencies (obtained from real experiments) are modeled as 7.5 ms and 5 ms, respectively.

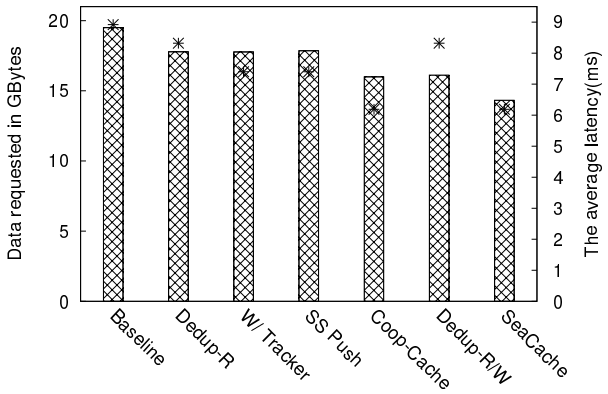

Figure 5.5 I/O bandwidth consumption and average latency under CIFS trace.

CIFS Trace In this experiment, we analyze how much data is transferred between the storage server and 72 hosts for all of the above mentioned algorithms using CIFS trace. The trace involves 717 clients reading 17.2 GB and writing 7.3 GB of data. Figure 5.5 shows the I/O bandwidth consumption (in GB) and the average latency of CIFS trace. Note that, we consider the amount of data that will be exchanged between the hosts and the storage server as the measure of bandwidth consumption, as reducing this data results in better utilization of the available I/O bandwidth.

Baseline performs worse than the other algorithms by consuming 19.5 GB bandwidth with average latency of 8.92 ms because it does not share CSI information. *Dedup-Read* reduces I/O bandwidth consumption by 8.7% and lowers the average latency down to 8.6 ms by eliminating duplicate read requests. For our two read path protocol variants, *W/ Tracker* and *SS-push*, the total I/O bandwidth reduction is about the same with *Dedup-Read*, while the latency is reduced down to 7.4 ms. This shows our two variants can effectively remove the extra I/O consumed by *Dedup-Read*.

Dedup-RW reduces I/O bandwidth consumption by 17.3%, while *Coop-Cache* and *SeaCache* reduce I/O bandwidth 17.9% and 26.5%, respectively. We see that *SeaCache* outperforms *Dedup-RW* by 9.2% in terms of bandwidth reduction and 25% in terms of latency. This is because *SeaCache* effectively optimizes the read path and redistributes the read path workloads to other physical hosts.

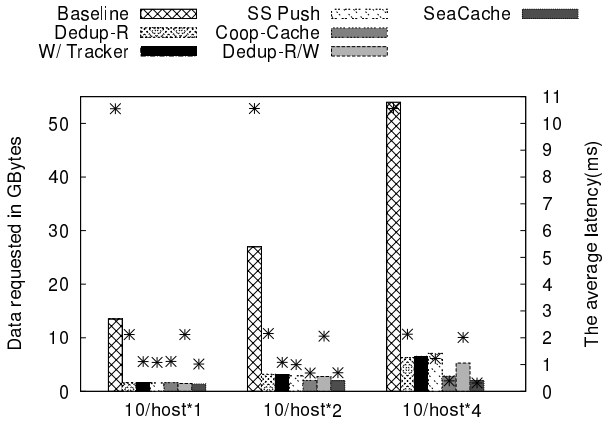

Figure 5.6 I/O bandwidth consumption and average latency under virus-scan storm.

Virus Scan in VDI Trace Figure 5.6 shows the I/O bandwidth consumption and average latency in VDI environment for a Virus-Scan Storm under different protocols and different number of VMs. The three groups of bars in the graph are: 10 VMs on a single host, 20 VMs on two hosts, and 40 VMs on 4 hosts. Within each group, the I/O bandwidth consumption and average latency per I/O are presented. It is observed that *SeaCache* is the best protocol compared with other six variants, as it achieves up to 96% I/O saving and 97% latency reduction compared with *Baseline*. The more clients are involved, the more benefits we can get from *SeaCache*. We can see that when 40 VMs are running, *SeaCache* outperforms *Dedup-RW* by 6% and 7% in corresponding I/O and latency reduction, respectively. Note that, even when the number of VMs running virus scan traces increases from 10 to 40, the I/O load seen by server increases by only 1.4 X under *SeaCache*, while *Baseline* saw the load increase by 4 X.

Two Weeks VDI Trace For this experiment, we traced the VDI environment for two weeks. Here since we have only 9 VMs involved, to show the effectiveness of *SeaCache*, we configure each physical machine to host at most 3 VMs. Thus, in this experiment, a total

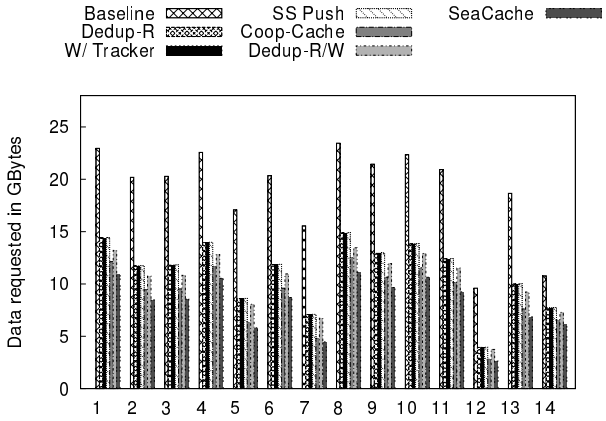

Figure 5.7 I/O consumption on persistent VDI traces for two weeks of usage.

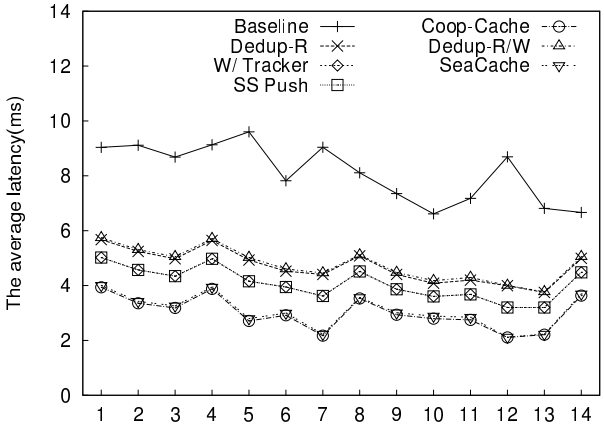

Figure 5.8 Average latency per I/O on persistent VDI traces for two weeks of usage.

of 3 physical hosts are used.

Figure 5.7 and Figure 5.8 show I/O bandwidth consumption and average latency, respectively, under different scenarios for the VDI trace. The x-axis here represents each day of the two weeks trace duration. *Baseline* perform significantly worse compared to any of the CSI sharing algorithms in terms of the amount of data transferred between hosts and storage server as well as the latency. This experiment shows that the performance of *SeaCache* consistently exceeds *Dedup-RW* by up to 14% in I/O saving and 24% in latency reduction.

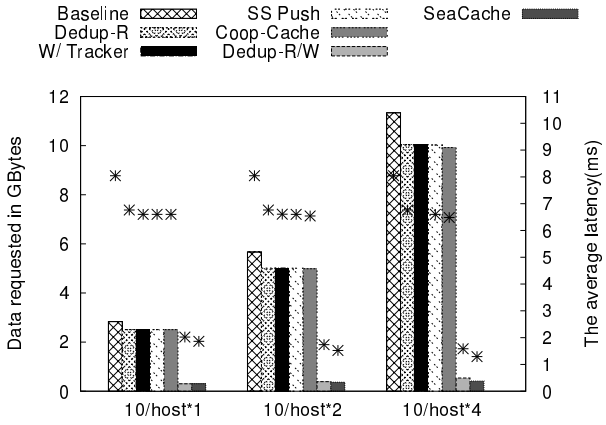

Figure 5.9 I/O bandwidth consumption and access latency under Test-Dev traces.

Test-Dev Trace Similar to Figure 5.6, Figure 5.9 shows the I/O bandwidth consumption and average latency under different algorithms for the test-dev trace. The *Dedup-R* and our read path optimizations do not gain significant benefits because the test-dev trace is write

intensive with write ratio of 86%. On the other hand *Dedup-R/W* and *SeaCache* can effectively reduce the duplicate writes. However, *SeaCache* does not win much over *Dedup-R/W* in this case since *SeaCache* focuses more on read path optimization (improving 1% in data saving and 3% in latency reduction). As in the previous experiment, increasing the number of VMs running the trace from 10 to 40 only results in a 1.8 X increase in the I/O load seen by the storage server, where as *Baseline* experienced a 4 X load increase.

The above experiments show that SeaCache can enable data center managers to provision storage servers for average loads instead of peak loads, and consequently improve the overall efficiency of the center.

5.3.2 Efficiency of Enhanced Host Cache

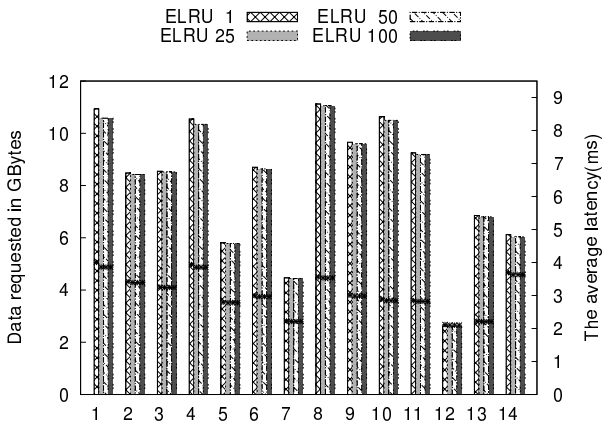

Figure 5.10 Enhanced LRU in host cache for persistent VDI traces.

In our next experiment, we use the simulator to test whether enhanced LRU (ELRU) algorithm that takes CSI information into account can perform better in comparison to basic LRU for host cache management. Figure 5.10 shows the results for the two weeks of VDI traces under *SeaCache*. Here, ELRU-n implies that the least shared entry within the last n LRU entries is evicted, e.g., ELRU-1 is the same as LRU. It is observed that ELRU-n performs slightly better than LRU; improving 1.5% in bandwidth saving and 1.5% in latency reduction. The main reason for this behavior is that LRU already accounts for recent accesses. Thus, if a shared block is being accessed multiple times by different VMs, it is not evicted as it is often not the least recently used block.

5.3.3 Storage Cache Efficiency

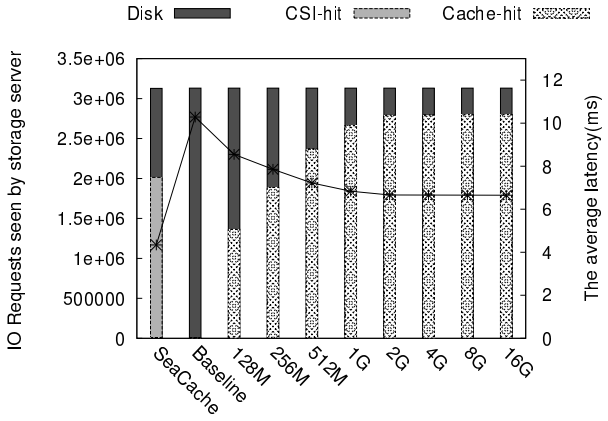

Figure 5.11 Impact of CSI on storage server cache for data blocks.

In this experiment, we compare the impact of adding more memory to the storage server for storing data blocks (without CSI sharing protocol) with a storage server that uses our CSI sharing protocol but does not use extra cache. Note that the I/Os shown here are for data that is not present in the client cache, which are sent to the storage server. For this test, we pick the 5^{th} day of VDI traces, as that yields decent performance under *W/ Tracker*. As shown in Figure 5.11, *W/ Tracker* is able to perform better against the one with the extra cache (as much as 16 GB) for mitigating disk I/O. This is because any CSI-Map hit causes the storage server to respond to the host with just CSI information. In contrast, although any storage server data cache hit eliminates disk I/O, it does not prevent the transmission of the larger payload to the host. The key insight here is that the CSI sharing protocols reduce the size of payload that needs to be serviced back by the storage server, which drastically reduces the average latency experienced by the hosts. This is an important result because it emphasizes that the capital spent on provisioning larger caches on the storage server can now be moved to the hosts. Finally, more hosts can be served using a single storage server, especially for workloads that are friendly to our protocols, such as VDI or Test-Dev environments.

5.3.4 Implementation Results

In this experiment, we use our prototype implementation to show that CSI sharing is a feasible idea. To this end, we traced the I/O requests of booting Windows XP-SP3 one after the other 100 s apart. Each boot of VM requests about 400 MB of data.

	Network Hit (%)	CSI-Map Cache Hit (%)	Average Latency (ms)
Baseline VM1	100.00	-	8.58
Baseline VM2	100.00	-	7.29
SS-Push VM1	99.00	01.00	8.97
SS-Push VM2	64.56	35.44	3.39

Table 5.1 Average latency per I/O on VM boot.

First, we compare the overhead introduced by CSI sharing in terms of average latency of booting the first VM for both *Baseline* and *SS-Push*. Table 5.1 shows that the overhead is negligible. Next, we observe the latency for booting the second VM. The CSI-Map cache hits drastically reduce the average latency of blocks requested on booting the second VM: about 35% of *SS-Push* requests for VM2 were identical to that for VM1, and these were cache hits in the CSI-Map. That shows that host can absorb the boot storm with out impacting the storage server, which would drastically reduce the design requirement on storage server from handling peak loads to average load.

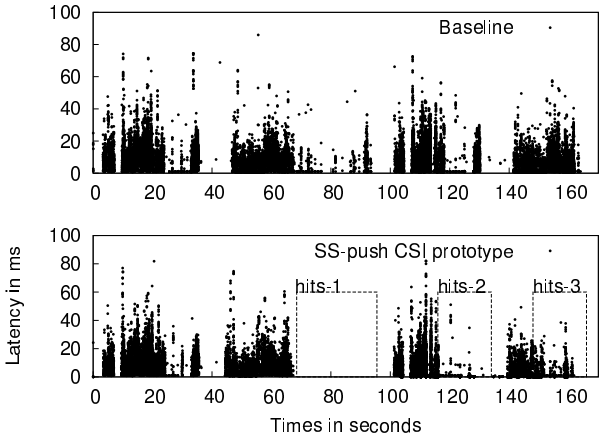

Figure 5.12 Latency of each I/O request on booting two VM's one after another.

Next, we measured the request latency for each I/O request as shown in Figure 5.12. We have marked three regions on the figure: hits-1, hits-2 and hits-3. Hits-1 presents the region where the *SS-Push* for VM1 re-requests blocks again post boot. These requests are absorbed by the CSI-Map cache and therefore we observe no latency in servicing these requests. The regions hits-2 and hits-3 mark requests by *SS-Push* for VM2 that has not been modified by VM1. We observed that the average latency for block-reads for VM2 dropped by 62%. With better CSI-Map between blocks and content, we expect to obtain a high rate of hits on a boot workload. This approach is very useful when tens of VM's are booted on a host, and the approach can mitigate boot-storms or virus-scan storms using a software-only solution.

5.4 Chapter Summary

In this Chapter, we presented an integrated approach, SeaCache, which incorporates host side caching, storage server to host data transfer and de-duplication information sharing protocols, and a storage server side de-duplication mechanism. SeaCache allows data center operators to (a) not provision resources for peak loads (for VDI type workloads) and (b) not procure extra hardware resources such as caches or on-wire de-duplication boxes. In this regard, we present: 1) algorithms for how storage server side de-duplication information can be leveraged to optimize storage server to host data transfers and host side caching; and 2) how host side client cache information can be leveraged at the storage servers to efficiently perform data transfer operations. We analyzed the proposed algorithms using three real-world workload traces and the results support our hypothesis that looking at these three system design areas in an integrated manner leads to overall bandwidth and latency benefits. Our experiments show that compared to dedup-box, SeaCache improves the I/O saving by up to 14% and latency reduction by up to 25%. Moreover, the results prove that SeaCache effectively absorbs the peak load under Virus-scan storm and Test-Dev traces. We have also developed a proof-of-concept implementation of SeaCache with an NFS client using a modified NFS protocol (that is CSI sharing protocol aware) and made the necessary changes at the NFS server to further validate that the approach is viable.

Chapter 6

I/O Similarity Aware Virtual Machine Management

As we mention in Chapter 5, scaling shared storage to support large VDE deployments poses issues of sustaining high performance while managing the high cost of provisioning large storage volumes. I/O reduction techniques, such as SeaCache, aiming at improving the performance and scalability of shared storage have been proposed based on the observation that virtual environments exhibit a large number of common data accesses between different VMs. However, without taking such similarity into account, VMs with similar I/O accesses may get placed on different physical hosts, reducing potential for I/O reduction and leading to suboptimal performance.

In this Chapter, we design SMIO, a VM placement system that monitors the I/O accesses of VMs, and places VMs with similar I/O accesses on the same physical host to improve the I/O reduction efficiency, which in turn helps improve the performance and scalability of shared storage system. We then evaluate the effectiveness of SMIO for read-intensive workloads and for read/write workloads compared to a system that does not employ similar-access-aware VM placement.

6.1 System Design

6.1.1 Design Rationale

The goal of SMIO is to efficiently detect I/O similarities among different VMs, cluster the VMs with similar workloads together and place the clustered VMs on the same physical host

if migration benefits exceed costs. The system needs to periodically re-adapt to the workload changes by creating better VM placements when possible.

A straw man approach is to use a centralized VM placement manager to periodically collect I/O access information from each VM. The I/O information is collected by installing an I/O monitoring component on each VM. The monitor computes the hash value of each block being accessed on the VM using collision resistant hash functions, such as SHA-1 [106], and periodically sends a list of collected hash values to the centralized placement manager. The manager uses the gathered information to cluster VMs with similar accesses, which can then be assigned to a physical machine (PM) together. Moreover, the manager can use the information to adapt to changes in workload characteristics.

While effective, the centralized approach is not scalable as the massive number of hash values from thousands of VMs being periodically sent to the manager will consume a large fraction of available network bandwidth and compete with the applications' communication, thus reducing overall system performance. Moreover, as the number of VMs increases, the bandwidth of centralized manager will easily become a bottleneck. To address the problem, we design a hierarchical approach, where individual hosts/VMs collect local I/O access information, process it, and only report a summary to the centralized manager. However, the approach still requires a central manager to collect the global information from all hosts to make proper VM clustering decisions. We design SMIO to address the above challenges. Specifically, the design goals of our system are as follows:

- Scalability: the VM placement manager should be able to quickly generate a solution for thousands of VMs, and the solution time should grow very slowly if at all.
- Low overhead: the work performed at hosts and VMs for collection and processing of the I/Os should minimally impact the performance of VMs.
- Low bandwidth consumption: the communication between the VM placement manager and the hosts should be minimal.
- Dynamicity: the VM placement manager should be able to dynamically reconfigure the placement topology to better adapt to the detected I/O workload changes.

6.1.2 Terminology

Here, we introduce the terminology that we have used.

- Cluster: A cluster consists of one or more VMs. (c_i, c_j) denotes a cluster merged from cluster c_i and cluster c_j.
- Cluster size (s_{c_i}): the number of VMs within cluster c_i
- λ_{c_i}: Num of unique blocks accessed by cluster c_i
- α_{c_i}: I/O similarity of cluster c_i, $\sum_{vm_j}^{vm_j \in c_i} \lambda_{vm_j} - \lambda_{c_i}$.
- β_{jk}: I/O saving of block b_k accessed by VM vm_j within cluster c_i, $1 - 1/$(the number of VMs accessing the block within cluster c_i).
- β_{vm_j}: I/O saving of VM vm_j, $\sum \beta_{jk}$ for all unique blocks accessed by VM vm_j within cluster c_i.
- Data sharing matrix M_{DS_k}: It is defined as

$$\begin{bmatrix} - & (\alpha_{12}) & \cdots & (\alpha_{1n}) \\ \vdots & & \ddots & \vdots \\ & \cdots & & (\alpha_{(n-1)n}) \end{bmatrix}$$

  , where n refers to the number of clusters. M_{DS_k} represents the $\alpha_{i,j}$ of cluster (i, j) under a distinct hash value range taken in charged by host k.
- Migration cost ($mcost(c, dst)$): the migration cost incurred by moving cluster c, to physical host dst.

6.1.3 System Overview

The architecture overview of SMIO is shown in Figure 6.1, where it runs on the Xen platform [111]. The targeted environment comprises of a shared storage system for persistent data storage and hosts organized in racks. The shared storage system eliminates the need to migrate the VM disk image files during migration, and only requires moving the in-memory VM state. Each host supports a number of VMs and has a distributed hash table (DHT) node running in the most privileged VM (Dom0). Each VM has an I/O monitor running in its guest OS. The I/O monitor traps the application I/O accesses at the block level, computes the hash values and sends it to the hosts corresponding DHT node periodically. Each DHT node is responsible for a distinct hash range. A VM placement manager runs on a dedicated

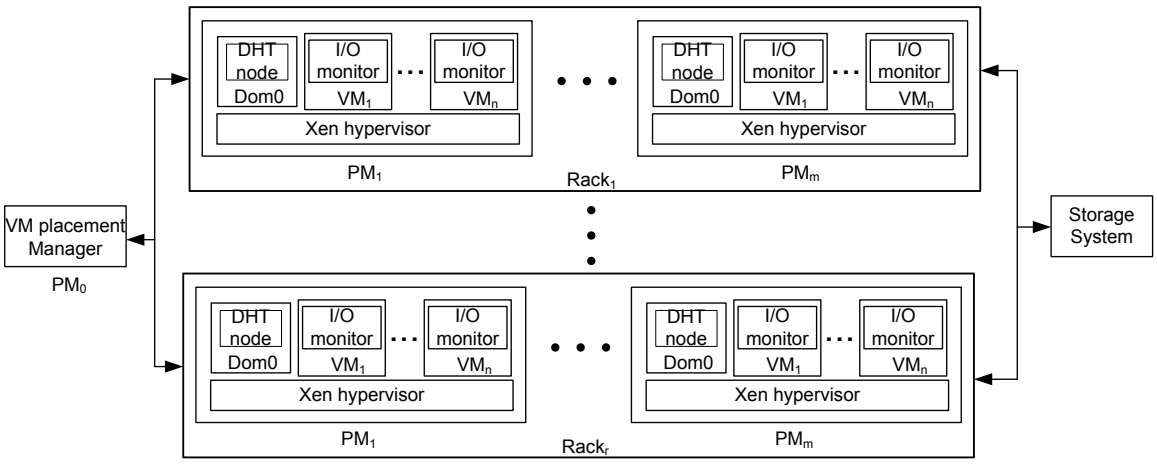

Figure 6.1 The overall system architecture.

host, which implements most of the intelligence of SMIO. The Xen hypervisor on every host receives instruction from VM placement manager for VM placement and migration. The VM placement manager collects information from DHT nodes in each host and uses hierarchical clustering (HC) [112] to generate a VM placement scheme. Hierarchical clustering is a widely used data analysis tool, which successively merges similar groups of points to create clusters of similar items. Compared with k-means [113] or k-medoids [114], hierarchical clustering does not require specification of the number of clusters k, which is an unknown in our environment. After the clustering scheme is generated, a migration execution algorithm is invoked to determine the actual placement of each cluster.

We adopt a bottom-up approach for hierarchical clustering. Each VM starts as a cluster with only itself as a member, then merges with other VMs (clusters) are performed successively until the algorithm can no longer find a suitable cluster to merge with based on the defined clustering criteria. The criteria factors in improved I/O similarity between the clusters to be merged and the cost of migrating the associated VMs. This is critical, as while merging clusters with high I/O similarity are preferred, the resulting migration overhead may negate the benefits. Such cases may arise, for example, when the two candidate clusters are far apart in terms of network distance. Once a suitable clustering plan is determined, a VM migration executor generates a migration plan aiming at minimizing the number of migrations, resulting network traffic, and migration time.

6.1.4 Net Benefit Modeling of Hierarchical Clustering

In order to cluster the VMs more effectively, SMIO defines the net benefits of a merged cluster c as shown in equation 6.1.

$$\gamma_c = S_{block} * (\alpha_c - \sum_{i}^{i \in c} \beta_{vm_i}) - a * mcost(c, DST) \tag{6.1}$$

S_{block} is the storage block size, a is a parameter used to adjust the weight between benefit and migration cost, β_{vm_i} is the I/O saving of vm_i within the cluster composing of VMs residing on the same physical host as vm_i. The first term $S_{block} * (\alpha_c - \sum_{i}^{i \in c} \beta_{vm_i})$ quantifies the improved similarity by merging two clusters and represents the number of block accesses can be saved in the current epoch if its member VMs are placed together compared with the I/O saving under current placement.

We purposely do not consider the frequency accessed by a single VM, instead we valued the frequency accessed by multiple VMs. The reason is that the former can benefit itself from mechanisms like SeaCache without the need to place similar VMs together. If each VM is placed separately, the unique blocks accessed by each VM will be requested from the storage server once. The subsequence accesses to an unique block from the same VM will be satisfied by the host cache. If VM 1 accesses block 1 at the first time, block 1 will be requested from storage. After that, the accesses to block 1 from VM 1 will hit the cache, but the accesses to block 1 from VM 2 will still go to storage since VM 2 is in a different host. Thus, the total accesses to the storage server from all separately placed VMs will be the sum of the unique block accesses from each VM. If the VMs in a cluster are placed together, only the unique blocks accessed by the cluster will be requested from the storage server once. The subsequence accesses to an unique block from the same VM or different VMs in this cluster will be satisfied by the host cache. For example, VM 1 is the first one in the cluster to access block 1, and the request of block 1 goes to the storage server. After that, the access to block 1 from VM 1 or other VMs in the cluster will not go to the storage server. Thus, the total accesses to the storage server from all VMs in the same host will be the total number of unique blocks accessed by the cluster. Therefore, the improved benefit is defined as the above.

Calculation of the migration cost needs careful consideration. Under a shared storage infrastructure that does not require migrating the VM disk image files, the migration cost of grouping two clusters mainly depends on the allocated memories of clusters and the network

distance between them. Network distance here refers to the hops required to transfer the in-memory data from one host to another. The larger the allocated memory clusters have and the longer the distance, the higher network traffic they would incur, thus leading to higher migration cost. The reason is that a typical live VM migration involves copying the memory pages from the source host to the destination host across the network.

In HC, the DST is *null* because we do not know which PM to place the cluster yet. We estimate the migration cost of grouping two clusters as the smaller of the allocated memories size among the two clusters times the network distance between the two clusters. The allocated memory size of a cluster is the sum of the allocated memory size of its children. The network distance is the minimum network distance between any child pairs from the two clusters. This is an optimistic estimation since the actual migration cost cannot be better than the migration cost estimated here. For example, consider that VM 1 with memory size of $1G$ and VM 2 with memory size of $2G$ have high similarity and the network distance is 1, i.e., they are located within the same rack. The migration cost in this case is $1G * 1 = 1G$. However, in reality, two other cases can happen. The migration executor might decide to move VM 2 to the host of VM 1 with a migration cost of $2G * 1 = 2G$, or both VM 1 and VM 2 are moved to some other host with network distance d resulting in a migration cost of $3G * d$.

In our system, we adopt a two-phase migration cost estimation approach. We estimate optimistically in hierarchical clustering to improve the quality of the cluster, which values both similarity and migration cost. For instance, consider if (VM 1, VM 2) have slightly better similarity than (VM 1, VM 3), but the migration cost of (VM 1, VM 2) is much higher than (VM 1, VM 3) due to far apart network distance or large allocated VM memory. It would be desirable to cluster (VM 1, VM 3) rather than cluster (VM 1, VM 2). Without the rough estimation in hierarchical clustering, it would be difficult to execute the migration plan in migration phase, and the possibilities for improve system performance are hampered. Later on in the migration execution phase, SMIO again estimates the migration cost more accurately and compares it with the similarity improvement achieved by migrating clustered VMs to a host together. The estimation of migration cost in the second phase has much higher accuracy because it has the global actual migration information. Further details of this are provided in Section 6.1.8.

Algorithm 2 The hierarchical clustering algorithm used in SMIO.

VM manager:
Epoch current_epoch;
On every t_1 minutes:
n=0;
while true **do**
  send to all DHT nodes: getDSMatrix(n, current_epoch, null);getBetaValue;
  Gather all M_{DS}, list of β_{vm} values from all DHT node;
  Calculate global data sharing matrix by $\sum M_{DS_i}$;
  Calculate and Sort the γ entries decreasingly;
  Group the cluster pairs which can fit into a physical host and have highest value of γ;
  if new schemes generated == false **then**
    break;
  end if
  n++;
  Broadcast the new scheme to all DHT nodes;
end while
Send the generated plan to VM migration executor.
Start a new epoch by increasing current_epoch by one and send to all DHT nodes;

DHT node:
Hashtable _ht_dht; List β_{vm}
Epoch current_epoch;
Onreceive getDSMatrix(n, current_epoch, null) from VM manager:
Send message M_{DS} to VM manager;
Onreceive getBetaValues from VM manager:
Calculate and send list of β_{vm} values to VM manager;
Onreceive the new scheme:
Update the M_{DS} for new cluster (i, j), delete column j, row j, update column i, row i, based on the definition.
Onreceive start new epoch e' from VM manager:
Clean up the hashtable _ht_dht and the data sharing matrix M_{DS}, list β_{vm};
Current_epoch=e';
Onreceive list of block hashes from a I/O monior:
Update the data sharing matrix M_{DS};
Merge the list of block hashes into _ht_dht;

I/O monitor:
Hashtable _ht_iotrace;
Sampling I/O accesses, store it in Hashtable _ht_iotrace;
On every t_2 sec, sends calculated block hashes to DHT nodes, cleans up _ht_iotrace;

6.1.5 Hierarchical Clustering in VM Manager

Note that some of the VMs might not have similarity with other VMs. In this case, other orthogonal placement algorithms [51, 53] can be used to determine the placement of these VMs since I/O similarity does not have an impact for such cases. Similarly, for VMs that are launched without any prior collected I/O information, such placement algorithms can be used to do the initial placement until the SMIO VM manager processes and suggests an I/O similarity-based clustering scheme. The detailed algorithm is illustrated in Algorithm 2.

In SMIO, the VM manager and each DHT node maintain an epoch number *current_epoch* as the local variable that is used to synchronize between VM manager and all DHT nodes. If a DHT node is out of synchronization, it is excluded in the current epoch. The DHT node's epoch number is then updated to join the next new scheme calculation. Every $t1$ minutes, the VM manager starts a new epoch by increasing the epoch number by 1 and launches a new round of hierarchical clustering.

Each DHT node maintains a hash table *_ht_dht* and a data sharing matrix M_{DS}. The keys in *_ht_dht* are block hashes (all falls in the specific range), and the value for each key is a list of VMs which accessed this block during the current epoch. Note that, the keys in the hash tables belonging to different DHTs do not have any overlap. It receives a list of hash values from I/O monitors periodically.

At the end of each epoch, the VM manager launches the hierarchical clustering algorithm to generate a new placement scheme. The clustering criteria shown in Equation 6.2 here is similar to the one defined at Section 6.1.4 yet adjusted to the hybrid distributed version.

$$\gamma_c = S_{block} * (\sum_{dht_k}^{dht_k \in DHT} (\alpha_c - \sum_{i}^{i \in c} \beta_{vm_i})) - a * mcost(c, null) \tag{6.2}$$

With the purpose of obtaining the criteria for the system, The VM manager first gathers data similarity matrices M_{DS} from all DHT nodes that have the same epoch number with iteration number 0. It then calculates the global data sharing matrix by summing up all the gathered data similarity matrices, followed by calculating the γ values of all cluster pairs sorting them in a decreasing order. Next the manager groups the cluster pairs into a new cluster, which can fit into a physical host and have γ greater than the threshold th_{sim}. The algorithm will not group two clusters together as a new cluster if the two clusters cannot fit into a physical host due to resource constraints. This makes sure that the cluster generated by the manager does not exceed the capacity of a physical host. If there are new

clusters generated, the VM manager will broadcast the new generated grouping scheme to all DHT nodes. Upon receiving a new grouping scheme, the DHT nodes update the M_{ds} for each new cluster (i, j) by deleting column j, row j, updated column i, row i as described in Section 6.1.7. The VM manager then proceeds to the next iteration for the current epoch.

If no more new clusters are generated, the VM manager sends the placement plan to VM migration executor, starts and sends out a new epoch number $current\_epoch + 1$ and terminates the current algorithm iteration. On receiving the new epoch number, DHT nodes clean up their hashtables and their data sharing matrices M_{DS}.

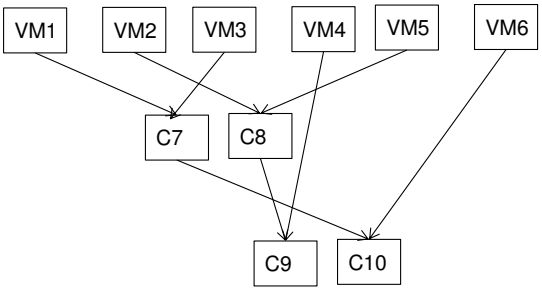

Figure 6.2 An example execution of hierarchical clustering.

Figure 6.2 shows an example execution of the hierarchical clustering algorithm. Initially, VM 1 to VM 6 each are separate clusters. After gathering the data sharing similarity matrices and calculating the global I/O similarity matrix, the VM manager determines to group $(VM1, VM3)$ and $(VM2, VM5)$ into new clusters $C7$ and $C8$, which have the first two highest I/O similarity ratio greater than threshold th_{sim}. $VM4$ and $VM6$ can not be paired because the I/O / similarity ratio is lower than th_{sim}. In the second iteration, the VM manager groups $C7, VM6$ into cluster $C10$, $C8, VM4$ intro cluster $C9$. In the third iteration, VM manager finishes the current algorithm. $C9$ and $C10$ cannot be further merged because no physical hosts can fit cluster $(C9, C10)$.

6.1.6 I/O Monitor

The I/O monitor inside each VM traps the block level data, calculates the hash value of sampled blocks, stores the block hashes as keys in its hashtable *_ht_iotrace*. The value for each hash key is simply *null*. We use a hash table to quickly identify the redundant I/O within a single VM and report unique blocks. Popular blocks within a VM should not affect

the similarities between VMs.

Every t_2 seconds, the I/O monitor periodically sends the hashes of unique blocks accessed by the VM during $(current - t_2, current]$, namely the collected hash keys, to corresponding DHT nodes. The keys are sent to the DHT nodes based on hash ranges and not directly to the VM manager. This distributes the network traffic across participating physical hosts, thus avoiding saturating the network bandwidth to the VM manager.

We implement the I/O monitoring process by modifying an existing I/O tracer for linux kernel, called blktrace, to also record the I/O content. The kernel space component of the tracer transfers the I/O events onto the userspace one, which among other things computes the fingerprint and passes that to the DHT node periodically. We would also like to mention that we chose to implement this in the VM itself, rather than doing in Dom0 as it was easy to modify an existing tool (blktrace). I/O monitoring and hash value computation can also be done in Dom0, which would be less intrusive to the VM users. That might also have lower overhead than in-the-VM approach used currently in SMIO.

In order to make I/O monitoring lightweight, SMIO samples the I/O accesses with uniform distribution [115] and only compute the hash value of sampled data blocks. The sampling significantly reduces the monitoring overhead in terms of CPU and memory utilization.

6.1.7 DHT Node Operation

DHT nodes work cooperatively with the VM manager to implement the hierarchical clustering algorithm. Each DHT node is in charge of a distinct range of block hash values. Assuming the block hash values are uniformly distributed, the work will be evenly distributed among the DHT nodes. DHT nodes help in offloading the computation and network traffic from the VM manager by grouping and summarizing the data sharing information between VMs before sending to the VM manager. This greatly increases the scalability of SMIO.

Each DHT node maintains a list of metadata info (L_m)of each VM including the λ value and β_{VM} value, both are 0 by default, the current VM placement topology information. The *_ht_dht* is initially empty at the beginning of each epoch. The data sharing matrix M_{DS} at each DHT node initially has 0 for all its α values in the matrix at the beginning of each epoch.

Then at every t_2 seconds, a DHT node receives a list of block hashes from an I/O monitor i

which monitors the accessed blocks in VM_i. While the DHT node merges the received block hashes into its own hash table *_ht_dht*, it also updates L_m and M_{DS}. The update algorithm is as below:

For each block hash *hash* from I/O monitor i (or VM_i), it looks up the hash table *_ht_dht*. If a block hash *hash* is already a key in *_ht_dht* and VM_i is not in the VM list, it adds the VM of this I/O monitor VM_i to the VM list for this key, increase α_i of L_m. For each VM j in this VM list, it also increases the number of common unique blocks α_{ij} (if $i < j$) or α_{ji} (if $i > j$) by 1. If *hash* is not a key in *_ht_dht*, it adds the key-value pair $(hash, VM_i)$ into *_ht_dht*.

On receiving getDSMatrix and getBetaValues, the DHT node iterate through the hash table, for each entry, iterate through the VM list, for each VM i, get its PM and the number of VMs n_{vm} which belonges to the same PM, $\beta_{vm_i} += 1 - 1/n_{vm}$, it sends VM manager the M_{DS} and L_m.

On receiving the newly generated grouping scheme from VM manager, assuming $i < j$ without loss of generality, for each newly merged cluster pairs (i, j), cluster i is updated to include all VMs previously belonging to cluster j and becomes new cluster i'. Cluster j is removed after getting the new cluster i'. The new cluster i' has the same column/row index as cluster i. This cluster to VM mapping information is also maintained in the VM manager.

In addition to updating the cluster-VM mapping, the data sharing matrix is also updated upon receiving the new grouping scheme from the VM manager. Row i in M_{DS} is updated as follows. For each cluster index k ($k > i$), walk through the hash table *_ht_dht* to get the number of common blocks α_{ijk} accessed by cluster i, cluster j and cluster k. The number of common blocks $\alpha_{i'k}$ accessed by cluster(i, j) and cluster k is calculated by $\alpha_{ik} + \alpha_{jk} - \alpha_{ijk}$. Column i is updated similarly for each cluster index k ($k < i$). After column i, row i are updated, column j, row j are deleted from matrix M_{DS}. As such, the updated matrix will be collected by VM manager for the next iteration.

6.1.8 Migration Execution

Once a new clustering scheme is generated, the migration executor is responsible for computing a migration plan. This plan specifies the host for each cluster. Since the hierarchical clustering may generate more clusters than the number of hosts, multiple clusters may share a single host.

One policy in the migration plan is to place an entire cluster rather than part of it in a host if possible since the purpose of this work is to put VMs with similar I/Os together. Under this condition, we try to minimize the migration cost. Computing a migration plan that minimizes the migration overhead is NP-hard because the well known NP-hard multi-dimensional knapsack problem can be reduced to it. Therefore, we design a greedy heuristic algorithm to determine the migration plan.

The main idea of this greedy algorithm is that initially each host has zero VMs or clusters assigned. Then the algorithm picks a cluster i and assigns it to a host j, based on the net benefit γ_{ij} of the cluster i can bring if the resource requirement does not exceed the physical limit. We next describe how to pick a cluster and a host in details. The definition of net benefit in Equation 6.3 is similar to the one described in Section 6.1.4.

$$\gamma_{ij} = S_{block} * (\alpha_{c_i} - \sum_{k}^{k \in c_i} \beta_{vm_k}) - a * mcost(c_i, pm_j) \tag{6.3}$$

For clusters with $s_c > 1$, $\gamma < 0$ means the benefit is less than the migration cost, the corresponding assignment is unqualified; if the cluster in its entirety is residing on host j, then $\gamma = 0$. Clusters with $s_c = 1$ have $\gamma = 0$ if they are on host j, $\gamma < 0$ if not. Next, we describe the algorithm in detail.

To decide which cluster to pick and the placement of it, the algorithm maintains a sorted list of γ values in decreasing order. Unqualified assignment with negative γ for clusters with cluster size greater than one is not in the sorted γ list. The algorithm picks the highest γ, assigns the corresponding cluster i to host j. The move is feasible because if host j does not have sufficient CPU, network, memory resources to fit cluster i, the γ value is $-\infty$. The affected γs are updated to reflect the placement decision just made. Specifically, the γ_{kj} are set to $-\infty$ thus removed from γ list if any unassigned clusters k can not fit host j. The γ_{ik}s of cluster i are deleted from γ list for each host k. The algorithm repeats the process until not positive γ values in the list. At this point, the unassigned clusters with cluster size greater than one will not be clustered. These clusters are then broken into clusters with size one, the corresponding γ values are updated and inserted into the γ list. The process is repeated until all clusters are placed.

After the new placement topology is generated, the migration executor needs to make sure that the new calculated placement topology actually outperforms the current placement topology before the actual migration. This is done by comparing the total benefit (amount of accesses saved) by changing from current placement to a new placement with the total

Workload	Training Center	Biobench1	Biobench2	TestDev
Similarity	medium	medium	varied	strong
Duration	2.5 hours	1.5 hours	46 min	36min
Read	28.9G	419G	407.7G	47.9G
write	30.5G	29G	10.2G	16.3G
# of requests	201K	495K	456K	4.1M
# of clients (VMs)	280	12	12	8

Table 6.1 Workload characteristics.

migration cost of this change. For the comparison, we compute μs for each physical host of new placement topology, the μ's for each physical host of current placement topology, the migration cost υs for each VM needed migration. The benefit-cost metrics Φ here is dfined in Equation 6.4.

$$\sum_{i \in P} (\mu_i - \mu_i') * S_{block} - a * \sum_{i \in Mset} \upsilon_i \tag{6.4}$$

In Equation refeq:phi, P is the set of physical hosts, $Mset$ is the set of VMs required migration. The cluster for μ_i of host i is the VMs assigned or residing on host i. If $\Phi > 0$, then the new calculated placement topology have a high possibility to yield better performance than the current solution after migration. Otherwise, the current migration plan is abandoned without changing the placement topology at this round.

The output of the algorithm is a list of clusters and the new destination host for each; the migration is triggered after the algorithm is terminated. Note that it is common to have clusters with cluster size one stay on the same host according to the computed migration plan.

6.2 Evaluation

We use trace driven simulations to evaluate the effectiveness of SMIO. In this section, we first describe the traces we collected, then we give details of the simulator, followed by a description of the experiments conducted.

6.2.1 Methodology

6.2.1.1 Workloads

We collected and used four different traces for our experiments, which are summarized in Table 6.1. The traces are classified into different similarity level, namely strong, medium and varied, based on the I/O accesses they exhibit. Strong similarity means different clients (VMs) have higher possibility to access the same data contents in a relatively small time frame, whereas varied similarity means different clients have a lower possibility to access the same data contents or access the same data contents at different time. Within the trace file, for each I/O access we collect the type of I/O (read/write command), the timestamp, the IP address of hosts, the file name, the offset and the size of the I/O and a list of hash values computed from actual data.

Training center traces: Running VMs within a training center is another common use case that presents similar I/O workloads. For example, in a TOEFL English Test training center, all the classes have the same time durations of typically 45 minutes. The first class usually begins at 8am with 4 classes packed in the morning. Within a class section, VMs owned by each student is likely to show similar I/O workloads. For example, VMs in a listening test section are going to retrieve the same audio file as students are instructed by the teacher to listen to a particular content. VMs in a speaking test section retrieve same spoken instructions but write different I/O contents to the shared storage as the audios recorded from different students are different. We collect the traces of a total of 280 students within 5 listening sections and 2 speaking sections. Each section comprises of 40 students. The VMs from the listening sections present strong similarity correlation, while the VMs from the speaking section show weak similarity correlation. The 7 sections begin at the same time with a total interval of 2.5 hours.

Bioinformatics benchmarking traces: These traces capture a typical scenario within scientific research centers such as national labs or university labs where users perform bioinformatics related research. Users in this case usually focus on the research of a particular DNA and protein and run search queries against corresponding databases. We use Blast [116] to demonstrate such a use case and collect the traces. Blast is a widely used DNA/protein sequence searching application. In our setup, three databases are used: NR with size of 17 G, NT with size of 14 G, and HTGS with size of 6 G. Our bio-benchmark 1 (biobench1) has 4 clients running on queries against each database with a total of 12 clients. Clients searching

against the same database run queries with different parameters representing the case that users are tweaking the parameters. Our bio-benchmark 2 (biobench2) has a similar setup except that clients searching against the same database run a set of 4 queries in different orders representing the cases that users are collaborating on research of same types of protein or nucleotide sequences. The similarities between VMs are varied in these traces.

Test and development traces: A typical scenario is an enterprise level test development environment where users usually continuously 1) develop/ edit codes, 2) compile codes, 3) install builds, and 4) conduct QA activities. Within an enterprise, different departments might be responsible for developing and testing different products and different teams within a department might be responsible for developing and testing different features of the same product. The group of VMs that test different features of the same product will typically exhibit strong similarities since the majority of the code base is the same. We setup such a test-dev environment and collect the traces. More specifically, there were four VMs for developing and testing Linux kernel version 2.6.32.15 and four VMs for developing and testing Xen 4.2. The 8 VMs read 47.9 G data and write 16.3 G data in total.

6.2.1.2 Simulator Design

Trace driven based simulation allows us to explore a variety of configuration spaces and the scalability of our system. We developed our simulator based on the one used in Chapter 5. Our simulator implements all the components shown in Figure 6.1 except the I/O monitor because we collected the traces offline. Particularly, the simulator consists of a DHT node, hierarchical clustering component and migration execution component. The hierarchical clustering component takes traces and configured system parameters as input and generates clustering schemes that are fed into the migration execution components. The migration execution component then computes the actual placement of clusters and executes the migration plan by instructing the SeaCache simulator to change the placement of VMs.

We assume each VM has the same cache size of 1 G, and the storage server cache size is 4 G. The parameters used in the following experiments are threshold th_{sim} of 0.2, decision interval of 90 seconds, and no sampling unless mentioned otherwise.

6.2.2 Effectiveness of SMIO

In our first set of experiments, we use our simulator to show the effectiveness of our system SMIO by comparing with First Fit Decreasing (FFD) [117] placement, the best and worst placement technique under test development, training center and biobench workloads. FFD is a greedy approximation algorithm designed for multi-dimensional bin packing problem, which attempts to place the VMs in the first host that can accommodate the VM. The order of hosts is sorted according to network architecture initially but fixed in all the algorithm runs. Particularly, hosts within a rack are neighbors in the host list. The placement of VMs is processed in the arrival order. The best/worst placement is the best/worst placement policy that yields the best/worst performance under different traces, which in general consumes the least/most I/O bandwidth between storage server and hosts. We obtained the best/worst placement manually to the best of our knowledge of the traces used.

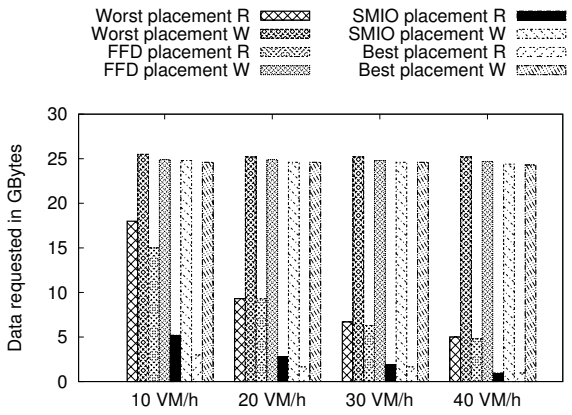

Figure 6.3 I/O bandwidth consumption under training center trace.

Training center trace Figure 6.3 shows how many data is transferred between storage server and 280 VMs for training data center trace under different number of VMs per host. The four groups of bars in the graph are 10 VMs/host with 28 hosts in total, 20 VMs/host with 14 hosts in total, 30 VMs/host with 10 hosts in total and 40 VMs/hosts with 7 hosts in total. With each group, the I/O bandwidth between hosts and storage servers are illustrated under FFD placement, SMIO placement and best placement policy. We observed that the different placement policies significantly impact the I/O bandwidth consumption between hosts and the storage server. As we can see, SMIO can effectively detect the similarity between VMs belonging to different sections and yields low I/O bandwidth consumption

comparable to best placement in all cases. On average, the read path performance of FFD is 4.9 times worse than SMIO while the write path is only 2.1% worse. The reason is that the students in each listening section of the training center trace listen to same materials most of the time during the section. This leads to high similarity of read workloads within each listening section. On the other hand, the students in the speaking section listen to instructions intermittently and record their speeches most of the time during the section, which results in nearly zero similarity for the write path I/O traffic. As the number of VMs per hosts increases, the I/O bandwidth consumptions are all reduced for all placement techniques.

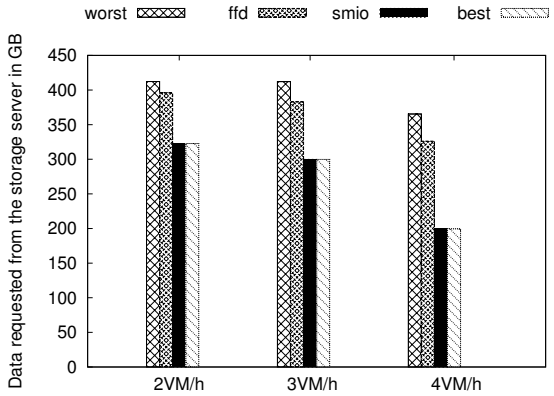

Figure 6.4 I/O bandwidth consumption under biobench1 trace.

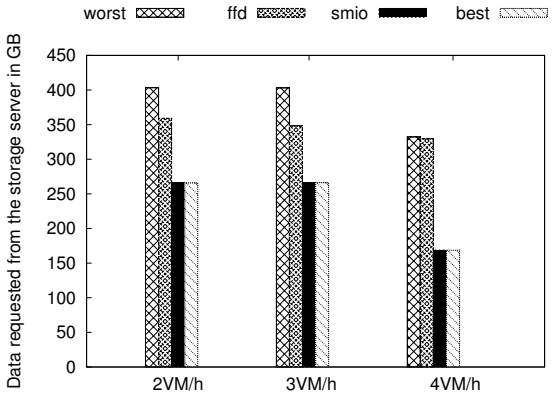

Figure 6.5 I/O bandwidth consumption under biobench2 trace.

Biobench traces Figure 6.4 and Figure 6.5 show the I/O bandwidth consumption for biobench1 and biobench2 traces under different placement policies and different number of VMs per host, namely 12 VMs distributed evenly on 3 hosts, 4 hosts and 6 hosts. In both traces, SMIO achieves almost the same I/O consumptions as the best placement policy. In biobench1, the I/O consumptions of the worst placement policy is 1.27, 1.37 and 1.87 times worse than SMIO for the three number of hosts considered while FFD placement is 1.22, 1.27 and 1.63 times worse than SMIO. In biobench2, the I/O consumption of the worst placement policy is 1.52, 1.51 and 1.97 times worse than SMIO for the considered scenarios while FFD placement is 1.35, 1.31 and 1.95 times worse than SMIO for the corresponding scenarios. It is observed that SMIO again effectively detects the similarity between VMs working on different data sets and groups the VMs correspondingly. Here we do not show the I/O consumption of read and write path separately because the traces are read-intensive with negligible write traffic.

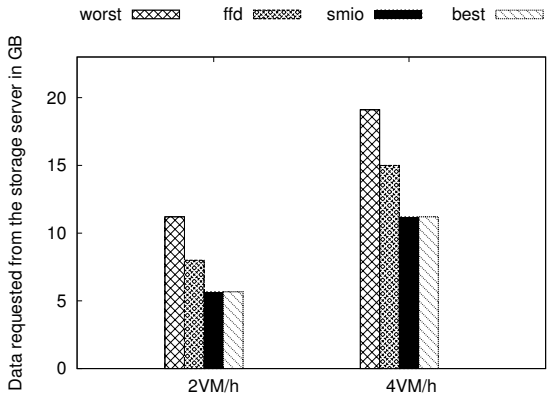

Figure 6.6 I/O bandwidth consumption under test development trace.

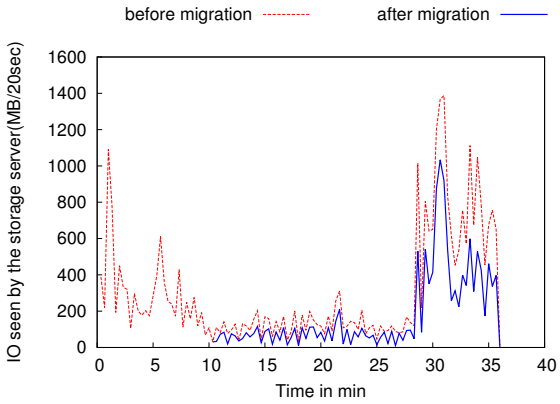

Figure 6.7 I/O bandwidth consumption under test development trace.

Test development trace Figure 6.6 shows similar results that SMIO effectively detects the I/O similarity and achieves the performance as the best placement. To see how the migration scheme is helping save the I/O consumption, we plot the I/O consumption over time using test development trace. In the simulator, we record the I/O and print out the value every 20 seconds. Figure 6.7 illustrates that after monitoring the I/Os for the first 5 minutes, SMIO decides to migrate VMs for new placement and the I/O seen by the storage server are consistently less than the I/O consumption before migration. Moreover, SMIO helps to reduce the peak bandwidth requirement by 33%. In total, SMIO reduces the I/O consumption by 74% compared to the base case.

6.2.3 Parameter Sensitivity Analysis

The performance of SMIO depends on the selection of various thresholds and system parameters, such as th_{sim}, t_1 and t_2, which need to be chosen carefully. Lower benefit-costs threshold th_{sim} indicates that more clusters will be paired up within each iteration, which results in fewer iterations and faster algorithm convergence. However, it may miss better pair-up opportunities. For example, cluster $3, 4$ might be paired up in n_{th} iteration with lower th_{sim}, which misses the opportunity to pair cluster 3 with cluster $1, 2$ in $n + 1_{th}$ iteration. On the other hand, if th_{sim} is set too high, it will increase the number of clusters with size 1 and deteriorate the quality of generated cluster scheme. In terms of t_1, if it is set too low, unnecessary resources will be wasted. To generate a new VM placement plan, network bandwidth will be consumed by the VM manager to communicate with DHT nodes not to mention the computing cycle and memory space utilized by Algorithm 2. If t_1 is set

too high, the potential I/O optimization opportunities will be missed. On the other hand, interval t_2 decides the length of the package delivered to corresponding DHT nodes. Small t_2 would lead to small packages with a larger network package header overhead, whereas large t_2 would lead to outsize packages, which would get large memory overhead since I/O monitors have to buffer them before sending to DHT nodes.

To quantify the quality of a generated cluster scheme, we use the Φ as discussed in Section 6.1.8. The experiments in this section are conducted with 40 VMs per host on 7 hosts. The results show a general guideline for picking th_{sim}, t_1 and t_2.

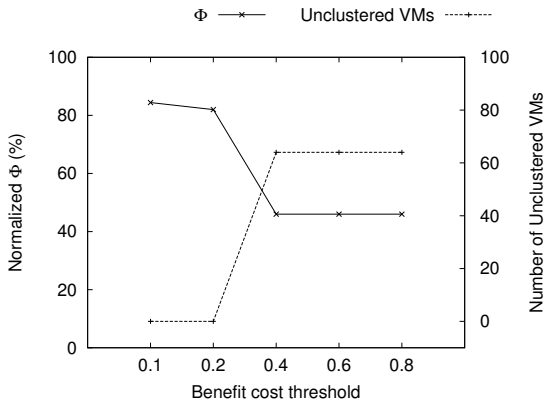

Figure 6.8 Impact of benefit-cost threshold under training center trace.

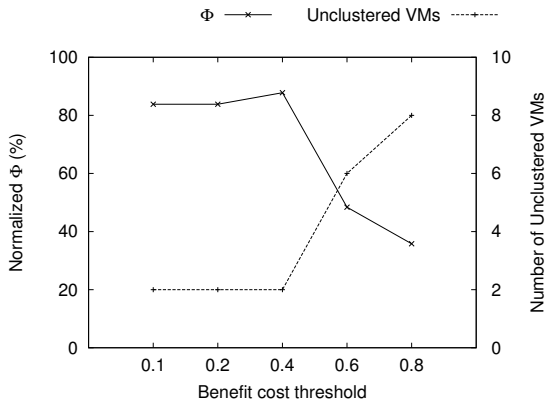

Figure 6.9 Impact of benefit-cost threshold under biobench2 trace.

6.2.3.1 Impact of Benefit-cost Threshold

This experiment explores the impact of benefit-cost threshold used in hierarchical clustering on the clustering result under both training center trace and biobench2 trace. In Figure 6.8, the x-axis is the benefit-cost threshold while the left y-axis is the Φ value. The higher Φ is, the higher is the quality of the generated clustering scheme. The right y-axis is the number of clusters with cluster size 1. We prefer this number to be low because such 1-sized clusters do not provide any useful information about the I/O similarity between VMs. We observe that as the threshold is increased, the similarity is decreased because the number of clusters with size 1 is increased from zero to 64 VMs. On the other hand, in Figure 6.9 Φ increases when the benefit-cost threshold is increased from 0.1 to 0.4 while the number of clusters with size 1 stays the same. However, when the benefit-cost threshold is increased from $0.4to0.8$, the Φ value actually goes down because the number of clusters with size 1 is increased from

2 to 8.

6.2.3.2 Impact of Decision Interval

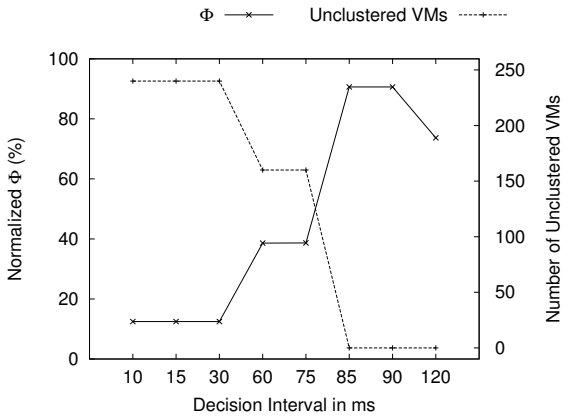

Figure 6.10 Decision interval and similarity score.

Next, we conduct an experiment to determine how different decision intervals would affect the similarity score and the number of clusters with size one. In Figure 6.10, we observe that as the decision interval is increased from 10 seconds to 85 seconds, the similarity score is increased because more block sharing information is obtained to help hierarchical clustering to make better clustering decision. However, the similarity score does not keep increasing and instead decreases as the decision interval is increased up to 120 seconds. The reason is that SMIO does not adapt to the dynamic workloads as soon as needed with a too long decision interval. In terms of the number of clusters with size 1, as the decision interval increases, SMIO can successfully cluster all the VMs.

6.2.3.3 Impact of Sampling Rate

The next experiment demonstrates the relationship between sampling rate of I/O monitor and the similarity score and the number of clusters with size 1 under hierarchical clustering. As expected, Figure 6.11 shows that the similarity score positively relates to the sampling rate. The higher sampling rate yields higher similarity score. This provides a trade-off for users to adjust. The sampling rate can be dynamically increased when the host has idle sys-

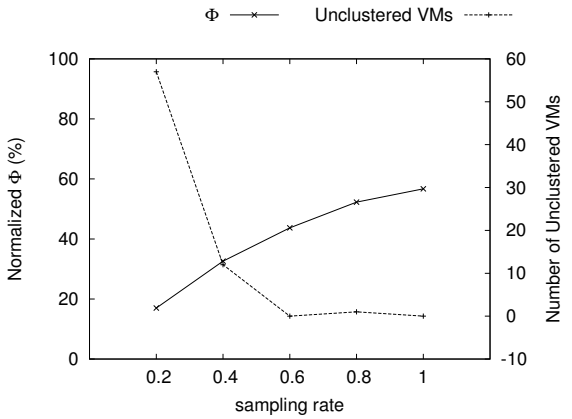

Figure 6.11 Sampling rate and similarity score.

tem resources and decreased when the VMs use up the system resources. On the other hand, the number of clusters with size one keeps decreasing as the sampling rate keeps increasing until to 0.6, which suggests a sweet configuration spot under this setup.

6.2.4 System Overhead and Scalability

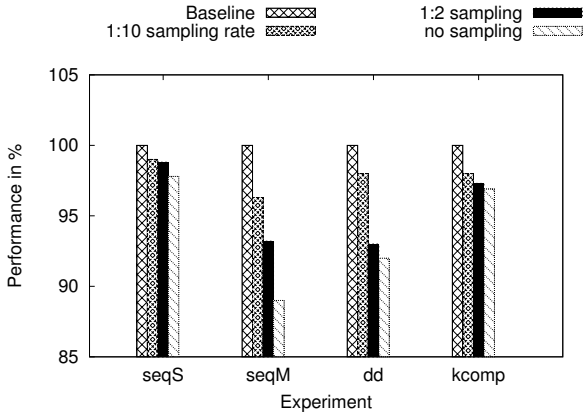

Figure 6.12 Overhead of the I/O Monitoring component.

Monitoring overhead: In this experiment, we measure the overhead of our I/O Monitoring component under different sampling rates, including without monitoring (baseline), 1 : 10 sampling rate, 1 : 2 sampling rate, and monitoring all I/Os (no sampling). The performance (speed) is normalized to the baseline performance. Since our monitoring component kicks in only when there are active I/Os happening to the disk, we test some I/O intensive and

I/O-CPU intensive applications. The results are shown in Figure 6.12. The first experiment is a sequential read of a 5 GB file (seqS) under various sampling rates. The experiment shows that even with no sampling, namely tracing all the I/Os, the observed read speed decreases by less than 3%. The same is not true for the multi-threaded (2 threads) version of this experiment (seqM), where we see how higher sampling rate helps to keep the performance hit in check. For our third experiment, we run the Unix utility `dd` to copy a 10 GB file. Again, as this is more "I/O-intensive" than the first experiment, we see a more decreased transfer speed (in terms of MB/s). Finally, we run the task of Linux kernel compilation, which is both computation and I/O intensive task. We observe that even with no sampling the observed read speed reduces by less than 4%. These results show that the overhead of the I/O monitoring can be kept low with an appropriate sampling rate; thus SMIO offers a feasible and practical approach to managing VMs.

We are concerned with three aspects of the algorithm, namely memory requirements, computation overhead, and network usage. We use the following symbols to represent various variables: N : total number of VMs; N_{pm} : number of VMs per physical machine; $M = N/N_{pm}$: total number of physical machines in the cluster; n_{uio} : total number unique I/Os received by a DHT node during an *epoch*; n_{io} : total number of I/Os received by a DHT Node during each *epoch*. Thus, we can roughly assume that total I/Os done by all the machines in the cluster n_{io}*M per epoch. As HC is a multi-round process, in which during each round C represents the total number of clusters (Intially this is N).

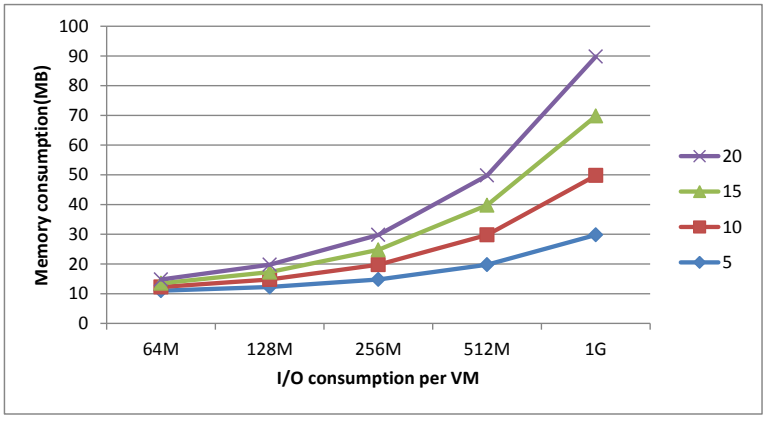

Figure 6.13 Memory consumption.

Memory requirements We store collected hashes (16 Bytes / hash) at each I/O monitor and send them to respective DHT nodes every t_2 seconds. $1M$ memory can easily track $256M$ I/Os. With a decent t_2 value, the memory consumption is negligible. At each DHT node, the main source of memory consumption is storing the hashtable which depends on N_{pm} and n_{io} value. Figure 6.13 shows the memory consumption within each DHT node as the value of n_{io} increases with the number of VM fixed at 1600. The higher N_{pm}, the more memory is required to store the hashtable. For $N_{pm} = 10$, capturing hash values representing $1G$ data of each VM as well as other data structures only requires $63M$ of memory size, which is acceptable for modern physical hosts which typically equipped with $32G$ or $48G$ memory. For the VM manager, the main source of memory consumption is storing the data sharing matrix and global net benefit matrix. When $N = 1600$, it requires around $20M$ to store the two matrices whose overhead is negligible.

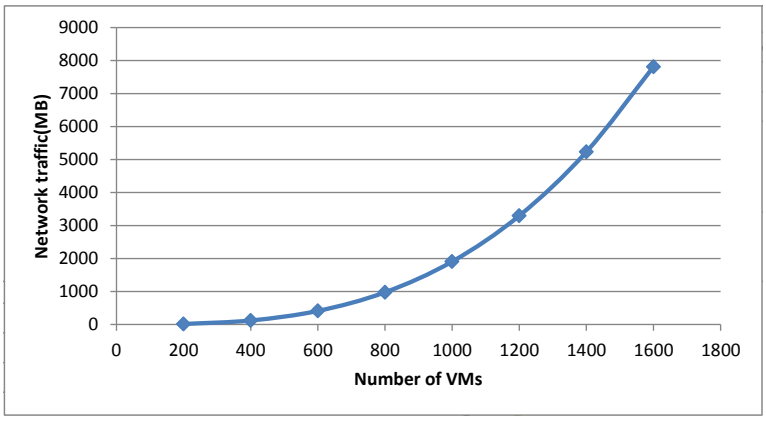

Figure 6.14 Network bandwidth consumption.

Bandwidth Usage Figure 6.14 shows the network bandwidth consumption of SMIO as the number of VMs increases up to 1600. Gathering the data sharing matrices from all the DHT nodes takes the largest portion of the consumption. For data centers up to 600 virtual machines, the algorithm consumes $411MB$ network bandwidth. For exceptionally large data centers with thousands of virtual machines, the matrix is typically observed sparse which considerably reduce the data amount transferred.

Scalability of SMIO The hierarchical clustering algorithm of SMIO takes no more than N_{pm} iteration to finish. For i_{th} iteration the size of the newly merged cluster (c_i, c_j) must be

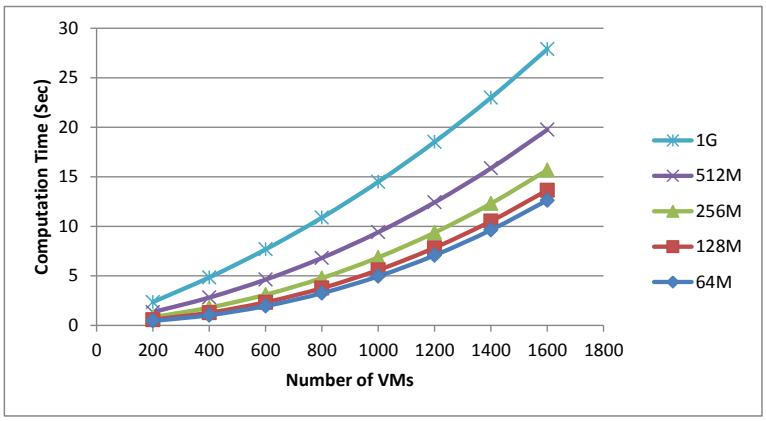

Figure 6.15 SMIO Scalability.

at least $i+1$. The reason is that either c_i or c_j must be merged at $i-1_{th}$ iteration. Otherwise, cluster (c_i, c_j) can be merged at $j_{th} < i_{th}$ iteration. That means, a path consists of merged clusters can be found from i_th iteration up to the first iteration for each cluster merged at i_{th} iteration. Within each iteration, the manager gathers the computed data sharing matrices from all DHT nodes, sorts the global matrices, generates the new clustering scheme which then is broadcast to DHT nodes. Upon received the new clustering scheme, each DHT node updates their local matrices accordingly. Figure 6.15 shows the algorithm completion time as the number of VMs grows up to 1600. For data center with up to 1600 VMs, $n_{io} = 1G/VM$, SMIO can finish within half minutes. That means SMIO is scalable.

6.3 Chapter Summary

I/O reduction is a valuable technique to improve storage system scalability in data centers, especially for VDEs with a shared underlying storage infrastructure. We observed that the similarity between VMs can improve the effectiveness of I/O reduction mechanisms, and the dynamic VDE workloads preclude static VM placement. In this context, we presented SMIO, a system that automatically performs I/O access monitoring, detects I/O similarities, quantifies the benefit and migration costs, calculates placement topology, and launches migrations that cluster VMs with similar I/O workloads on the same/nearby hosts. SMIO realizes: 1) a hierarchical clustering algorithm that clusters the VMs with similar I/O accesses in a hybrid distributed manner; 2) a greedy migration algorithm that computes the new mapping between VMs and hosts with the goal of eliminating unqualified clusters and minimizing the migration overhead; and 3) a two phase benefit-migration throttling tech-

nique that substantially improves the clustering quality, hence effectively improves the I/O savings between hosts and storage system in VDEs. A trace driven simulation showed that SMIO can effectively detect I/O similarities between VMs in a timely fashion, determine the migration plan and improve the storage system performance. Compared with the widely used FFD placement policy, SMIO can improve the performance of storage system by as much as 80% for read-intensive workloads and 50% for read/write workloads.

Chapter 7

Conclusion

The dissertation presents the design of a resource management framework for cloud computing. This framework targets two typical and widely used cloud services: MapReduce in the cloud and virtual desktop environment. For the cloud service of MapReduce in the cloud, we tackle the challenges and targeted problems both from cloud providers' perspective and users' perspective. Specifically, we introduce a topology aware min-cost-flow based resource manager to improve the MapReduce job performance for cloud providers. We also design a mrOnline system allowing users' to ease the burden of issuing accurate resource requests through parameter configuration. The use of presented framework can help improve cluster utilization and application performance. Moreover, the framework enables I/O bandwidth deduplication between centralized storage server and physical hosts to enhance storage scalability of VDE. Furthermore, it provides I/O similarity aware VM placement and migration to guarantee the I/O reduction efficiency.

Our topology-aware min-cost flow based resource manager manages data / VM placement for MapReduce in multi-tenant virtualized clouds. CAM relies on a three-level approach to avoid the placement anomalies because of overlaid topology and inefficient resource allocation. More specially, CAM exposes compute, storage and network topologies to MapReduce job scheduler, places data according to network traffic of corresponding jobs, expected machine load and storage utilization and places VMs with a goal of maximizing global data locality and job throughput. It reduces the network traffic of multiple MapReduce instances in a multi-tenant environment in which jobs are exhibiting different job characteristics. Thus, the application performance is considerably improved compared with a start-of-the-art MapReduce resource scheduler.

From the users' perspective, we also studies how to automate the parameter configuration eliminating users' responsibility to configure per job parameters and request accurate needed resources from MapReduce job scheduler. Parameter configuration is difficult due to a large

configuration space and dependencies on job characteristics, input data sets and cluster configurations. Traditional offline tuning is time consuming since it requires many test runs to generate a desirable configuration. Moreover, they cannot avoid cluster hotspots. mrOnline is offered to address these challenges. One key contribution which lays the foundation of mrOnline is enabling task level dynamic configuration. It allows to change configurations per task while the jobs are running. This key system-level improvement leads to huge opportunities to: (i) continuously tunes performance within a single run; (ii) reduces the number of test runs compared to offline tuning; and (iii) improves MapReduce cluster utilization when executing multiple jobs concurrently. mrOnline leans on a gray-box hill climbing algorithm to find a near-optimal configuration through the large configuration space. Moreover, mrOnline exploits MapReduce runtime statistics and incorporates tuning rules for key parameters using aggress and conservative strategies. The evaluation of mrOnline using the two strategies demonstrates that mrOnline can help automate the parameter tuning dynamically and in a much more efficient way.

For the second cloud service, we focus on investigating effective caching techniques for improving the storage scalability of centralized storage servers. Based on the observation that lots of duplicate data exist within VDEs, we introduce a holistic caching system, SeaCache. It consists of a host content addressable cache, storage deduplication and a content sharing protocol. We also observe the following results through our evaluation. Firstly, unlike current deployments, where virtualized environments are provisioned for peak loads in order to deal with boot storms (e.g. VDI environments) by the customers, SeaCache allows provisioning for average loads. Secondly, Many solution providers expect their customers to increase the size of the caches either at the hosts or the storage server in order to deal with peak workloads. SeaCache allows customers to give away with these cache extensions, thus providing for higher system efficiency. Finally, SeaCache algorithms are more efficient than simple on-wire data transfer solutions, where de-duplication boxes are placed at both source and destination ends to de-duplicate data being transferred across the wire.

The goal of I/O similarity aware VM management is to improve the efficiency of I/O reduction techniques including SeaCache. We notice that the effectiveness of I/O reduction methods depends on the I/O similarity of VMs running on the same physical hosts. The higher I/O similarity, the higher I/O reduction efficiency. We observe that VMs can exhibit cluster behavior in terms of I/O similarity. Thus, we propose SMIO which monitors and detects I/O similarities between VMs, employing hierarchical clustering algorithm to cluster VMS and place or migrate VMs based on the estimated benefits and migration costs.

The trace driven simulation shows that SMIO can effectively detect I/O similarities between VMs, decide the migration plan and improve I/O reduction efficiency.

7.1 Future Research

This is a cloud era in which resource management techniques serve as a fundamental enabling technology. In this dissertation, we have addressed the challenges of managing resources and enhanced storage scalability of two focused cloud services. 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 cloud computing and distributed systems.

7.1.1 Application-attuned Heterogeneous-aware Resource Management in the Cloud

Large distributed software framework (DSF) deployments such as MapReduce, Pig and Hive, in the cloud continue to grow in both size and numbers, given the DSFs are cost-effective and easy to deploy. However, a problem posed by modern 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.

Another problem faced in evolving DSFs is how to handle increasing heterogeneity in the underlying infrastructure efficiently. 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 such as buffers and SSDs in addition to disks.

Cloud providers usually construct cloud consolidation environment from a variety of machine classes as the generations of machines evolve overtime with more attractive cost-performance specifications [12]. Some machines might be even equipped with additional hardware such as

specialized GPU accelerators [118] or SSDs [119]. In a multi-tenant cloud hosting varieties of workload ranging from Web applications, databases to MapReduce [120] workloads, the resource demands from different organizations exhibit a high degree of heterogeneity. The heterogeneity from machine types and workloads significantly complicates the resource allocation and management in order to ensure high resource usage efficiency without waiting for a collection of resources allocated at the same time. Thirdly, the fact that workloads might have affinity or constraints for resources further sophisticates the resource management decisions. Some applications might require VMs with GPU accelerators. Some applications specifying VMs with 4GHZ CPU cores can not be allocated to machines equipped with 3GHZ CPU cores. The affinity constraints limits the set of physical machines that VMs can be migrated to. These factors are not captured well in the current popular virtual infrastructure managers.

To address the above problems, the following critical challenges have to be tackled. Data analytic 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 straggler detection, and data replica placement are not capable of exploiting heterogeneity in both the system architecture (e.g., different CPUs, embedded devices, GPUs, tiered-storage) and various stages of a workflow. Second, 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. Third, 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.

In this dissertation, I have studied how to allocate and manage cluster resources effectively to alleviate the placement anomalies for MapReduce instances running in the cloud. It is useful to extend the research to design an automated, cross layer performance optimization framework for DSFs which will be able to adapt to varying application and heterogeneous infrastructure characteristics at runtime to better drive resource management and thus achieving high performance and efficiency.

7.1.2 Storage Substrate Optimization for Cloud Infrastructure

While virtualization is a leading enabler for cloud computing especially for PaaS, the key challenge in this domain is to address the unpredictability of I/O virtualization due to high disk bandwidth deviations caused by I/O contentions. On the other hand, SSDs, exhibiting superior random I/O performance, are increasingly deployed in high-end storage systems such as high throughput key value stores. The advancement of SSDs has opened up new opportunities to improve the performance isolation in I/O virtualization. In my graduate research, I have studied the caching protocols de-duplicating I/O accesses to improve storage efficiency in virtualized environments. Here, I am interested in studying how SSDs can be utilized to deliver stronger performance isolation to avoid interference observed in traditional hard disks. I am also interested in investigating how the improvement of performance isolation facilitates the tight time coordination demanded by scientific applications in high performance computing.

As different DSFs keep emerging, it is interesting to explore how to dynamically share cluster resources between different DSFs and the corresponding data storage. This will facilitate data sharing and significantly improve cluster utilization compared with static partitioning. There are numerous remaining challenges. For example, how to efficiently manage caching tiers of DSFs hosting applications with different workload characteristics and service level requirements. Web applications and ad-hoc queries on non-SQL databases are latency-sensitive and interactive applications, which cache small objects. In contrast, MapReduce, like other batching applications, prefers all or nothing caching policies. I am particularly interested in studying how to allocate memory for multiple applications that belong to different DSFs and to build a cluster memory manager coordinating caching tiers.

Bibliography

[1] http://en.wikipedia.org/wiki/Cloud_computing. Cloud computing. Retrieved at Jan 2013.

[2] William Voorsluys, James Broberg, and Rajkumar Buyya. *Cloud Computing: Principles and Paradigms*. Wiley, 2011.

[3] http://aws.amazon.com/. Amazon web services. Retrieved at Jan 2013.

[4] https://cloud.google.com/index. Google cloud platform. Retrieved at Jan 2013.

[5] http://www.microsoft.com/OEM/en/products/other/Pages/cloud_services.aspx. Microsoft cloud services. Retrieved at Jan 2013.

[6] http://www.windowsazure.com/. Microsoft windows azure. Retrieved at Jan 2013.

[7] http://www.salesforce.com/. The salesforce cloud. Retrieved at Jan 2013.

[8] https://www.rackspace.com/. The rackspace cloud. Retrieved at Jan 2013.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[10] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[11] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 15:1–15:12, New York, NY, USA, 2012. ACM.

[12] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[13] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14 –22, sept.-oct. 2009.

[14] `http://www.amazon.com/b?ie=UTF8&node=201590011`. Amazon Elastic Compute Cloud (Amazon EC2).

[15] SATRAN J. Internet small computer systems interface (iscsi).

[16] K. Malavalli and B. Stovhase. Distributed computing with fibre channel fabric. In *Compcon Spring '92. Thirty-Seventh IEEE Computer Society International Conference, Digest of Papers.*, pages 269 –274, feb 1992.

[17] Rajagopal Ananthanarayanan, Karan Gupta, Prashant P, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari. Cloud analytics: Do we really need to reinvent the storage stack?

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, number 5, pages 29–43. ACM, 2003.

[19] Apache Software Foundation. Apache hadoop 2.2.0, 2014. http://hadoop.apache.org/docs/r2.2.0/.

[20] Apache Software Foundation. Grep example, 2014. http://wiki.apache.org/hadoop/Grep.

[21] https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/ package summary.html. Apache software foundation: Terasort example, 2014.

[22] Cloudera. 7 tips for improving mapreduce performance, 2009. http://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/.

[23] Cloudera. Optimizing mapreduce job performance, 2012. http://www.slideshare.net/cloudera/mr-perf.

[24] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

[25] Impetus. Hadoop performance tuning, 2012. https://hadoop-toolkit.googlecode.com/files/White paper-HadoopPerformanceTuning.pdf.

[26] Impetus. Advanced hadoop tuning and optimizations, 2009. http://www.slideshare.net/ImpetusInfo/ppt-on-advanced-hadoop-tuning-n-optimisation.

[27] 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*, volume 11, pages 261–272, 2011.

[28] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[29] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: recapitulating storage for virtual desktops. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[30] Min Li, Shravan Gaonkar, Ali R. Butt, Deepak Kenchammana, and Kaladhar Voruganti. Cooperative storage-level de-duplication for i/o reduction in virtualized data centers. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, aug 2012.

[31] http://www.unidesk.com. Unidesk: Virtual desktop management, 2009.

[32] http://www.riverbed.com. Riverbed steelhead family overview, June 2005.

[33] http://www.atlantiscomputing.com/. Atlantis computing, 2006.

[34] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[41] 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 *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[42] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50, 1508300. ACM.

[43] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *(INFOCOM)Proceedings of the 29th conference on Information communications*, pages 1154–1162, 1833690. IEEE Press.

[44] A. Kochut and K. Beaty. On strategies for dynamic resource management in virtualized server environments. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS '07. 15th International Symposium on*, pages 193–200.

[45] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. Autonomic virtual resource management for service hosting platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 1–8, 1564622. IEEE Computer Society.

[46] A. Kochut. On impact of dynamic virtual machine reallocation on data center efficiency. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–8.

[47] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128. min: reduce sla violation and the num of phyical host.

[48] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings*

of the 2009 conference on USENIX Annual technical conference, USENIX'09, pages 28–28, Berkeley, CA, USA, 2009. USENIX Association.

[49] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[50] S. Mehta and A. Neogi. Recon: A tool to recommend dynamic server consolidation in multi-cluster data centers. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 363 –370, april 2008.

[51] Xiaoqiao Meng, V. Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM, 2010 Proceedings IEEE*, pages 1 –9, march 2010.

[52] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra. Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 228 –237, sept. 2010.

[53] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

[54] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. of the VLDB Endowment*, 4(11):1111–1122, 2011.

[55] Herodotos Herodotou, Fei Dong, and Shivnath Babu. Mapreduce programming and cost-based optimization? crossing this chasm with starfish. *Proceedings of the VLDB Endowment*, 4(12), 2011.

[56] Guangdeng Liao, Kushal Datta, and Theodore L Willke. Gunther: Search-based autotuning of mapreduce. In *Proceedings of European Conference on Parallel Computing (Euro-Par 2013)*, pages 406–419. Springer, 2013.

[57] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[58] Eaman Jahani, Michael J Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.

[59] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[60] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. Perfxplain: debugging mapreduce job performance. *Proceedings of the VLDB Endowment*, 5(7):598–609, 2012.

[61] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[62] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y Li, Wei Lin, Jingren Zhou, and Lidong Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 22–22. USENIX Association, 2012.

[63] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[64] Apache. Mumak: Map-reduce simulator, 2009. https://issues.apache.org/jira/browse/MAPREDUCE-728.

[65] Tao Ye and Shivkumar Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):196–205, 2003.

[66] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 287–296, New York, NY, USA, 2004. ACM.

[67] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[68] Wei Zheng, Ricardo Bianchini, G John Janakiraman, Jose Renato Santos, and Yoshio Turner. Justrunit: Experiment-based management of virtualized data centers. In *Proc. USENIX Annual technical conference*, 2009.

[69] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 7, Berkeley, CA, USA, 2002. USENIX Association.

[70] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *ATC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[71] Alex Osuna and Rucel F. Javier. *IBM System Storage N series Software Guide*. IBM Redbook, July 2010. SG24-7129-04.

[72] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[73] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 111–123, Berkeley, CA, USA, 2009. USENIX Association.

[74] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[75] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking Back References in a Write-Anywhere File System. In *FAST'10: Proceedings of the 9th conference on USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2010. USENIX Association.

[76] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[77] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[78] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. In *International Symposium on Computer Architecture*, 2004.

[79] Niraj Toila, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Bressoud Thomas, and Adrian Perrig. Opportunistic use of content addressable storage for distributed file sytems. In *ATC 2003: Proceedings of USENIX Annual Technical Conference*, 2003.

[80] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proceedings of USENIX Annual Technical Conference*, 2000.

[81] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.

[82] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, New York, NY, USA, 2001. ACM.

[83] Navendu Jain, Mike Dahlin, and Renu Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proc. 4th USENIX Conference on File and Storage Technologies*, pages 21–21, Berkeley, CA, USA, 2005. USENIX Association.

[84] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: recapitulating storage for virtual desktops. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[85] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[86] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[87] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *Proceedings of USENIX Annual Technical Conference*, 2009.

[88] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. In *Proceedings of the 8th conference on file and storage technologies (FAST '10)*, 2010.

[89] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[90] Song Jiang, Fabrizio Petrini, Xiaoning Ding, and Xiaodong Zhang. A locality-aware cooperative cache management protocol to improve network file system performance. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, pages 42–, Washington, DC, USA, 2006. IEEE Computer Society.

[91] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005.

[92] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, and D. Subhraveti. Gpfs-snc: An enterprise storage framework for virtual-machine clouds. *IBM Journal of Research and Development*, 55(6):2:1 –2:10, nov.-dec. 2011.

[93] IBM. Ibm service agility accelerator for cloud documentation.

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

[95] A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996 –1005, march 2010.

[96] Apache Software Foundation. Hadoop, May 2007. `http://hadoop.apache.org/core/`.

[97] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[98] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[99] Apache. Apache giraph, 2013. http://giraph.apache.org/.

[100] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[101] Apache Incubator. Spark:lightning-fast cluster computing, 2013. http://spark.incubator.apache.org/.

[102] Twitter. Storm: Distributed and fault-tolerant realtime computation, 2013. http://storm-project.net/.

[103] Jimmy Lin and Chris Dyer. Cloud9: A hadoop toolkit for working with big data, 2010. http://lintool.github.io/Cloud9/index.html.

[104] Wikipedia. Wikipedia data dumps, 2014. http://dumps.wikimedia.org/enwiki/latest/.

[105] Google Inc. Freebase data dumps, 2013. https://developers.google.com/freebase/data.

[106] FIPS 180-1. *Secure Hash Standard*. U.S. N.I.S.T. National Technical Information Service, Springfield, VA, April 1995.

[107] Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the i/o bottleneck: network storage with trapeze/myrinet. In *Proc. USENIX Annual Technical Conference*, 1998.

[108] Steve Jin. *VMware VI and vSphere SDK: Managing the VMware Infrastructure and vSphere*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[109] Troy D. Hanson. Uthash: A hashtable for c structures, http://uthash.sourceforge.net/, September 2010.

[110] http://www.opendedup.org. Open dedup, September 2010.

[111] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[112] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[113] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[114] L.K.P.J. ROUSSEEUW. Clustering by means of medoids. *Statistical data analysis based on the L1-norm and related methods*, page 405, 1987.

[115] George P. Moore, David S.and McCabe. *Introduction to the practice of statistics*. W.H. Freeman Company, February 2005.

[116] http://blast.ncbi.nlm.nih.gov/Blast.cgi. Blast:basic local alignment search tool, November 2009.

[117] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[118] Peer1 Hosting. GPU Cloud Computing, July 2010. `http://www.peer1.com/hosting/gpu-cloud.php`.

[119] Rini T. Kaushik, Ludmila Cherkasova, Roy Campbell, and Klara Nahrstedt. Lightning: self-adaptive, energy-conserving, multi-zoned, commodity green cloud storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 332–335, New York, NY, USA, 2010. ACM.

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