The Pennsylvania State University
The Graduate School

TOWARDS IMPROVING PERFORMANCE AND RELIABILITY OF CLOUD PLATFORMS

A Dissertation in
Computer Science and Engineering
by
Bikash Sharma

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2013

The dissertation of Bikash Sharma was reviewed and approved* by the following:

Chita R. Das
Distinguished Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Mahmut T. Kandemir
Professor of Computer Science and Engineering

Bhuvan Urgaonkar
Professor of Computer Science and Engineering

Qian Wang
Professor of Mechanical Engineering Department

Joseph L. Hellerstein
Manager, Google Inc.
Special Member

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Cloud computing refers to both applications delivered as services over the Internet, as well as the hardware and system software in the data centers, that provides these services. It has emerged as one of the most versatile forms of utility computing, where both applications and infrastructure facilities can be leased from an infinite pool of computing resources in the form of fine-grained pay-as-you-go mode of billing. Over recent years, there has been an unprecedented growth of cloud services from various key cloud providers like Amazon, Google, IBM and Microsoft. Some of the unique characteristics that make cloud computing so attractive and different from traditional distributed systems like data centers and grids include self-organization, elasticity, multi-tenancy, infinite resource pool availability and flexible economy of scale.

With all the promises and benefits offered by cloud computing, also comes the associated challenges. Amidst various challenges, performance and reliability related issues in clouds are very critical and form the main focus of this dissertation. Specifically, the focus of this dissertation is to address the following four specific performance and reliability concerns in clouds: (i) lack of representative cloud workloads and performance benchmarking that are essential to evaluate and assess the various characteristics of cloud systems; (ii) poor management of resources in big data cloud clusters running representative large scale data processing applications like Hadoop MapReduce. Specifically, this is due to the problems associated with the static, fixed-size, coarse-grained, and uncoordinated resource allocation in Hadoop framework; (iii) inefficient scheduling and resource management of representative workload mix of interactive and batch applications, running on hybrid data centers which consist of both native and virtual machines; and (iv) lack of end-to-end effective problem determination and diagnosis framework for virtualized cloud platforms that is quintessential to enhance the reliability of the cloud infrastructure and hosted services.

Towards this pursuit, this dissertation addresses the above mentioned performance and reliability specific problems in clouds, explores the underlying motivations, proposes effective methodologies and solutions, conducts exhaustive evaluations through comprehensive experimental and empirical analyses, and lays foundations for future research directions. The first chapter of the dissertation focuses on the characterization and modeling of cloud workloads. In particular, the thrust is on the modeling and synthesis of an important workload property called *task placement constraints*, and demonstrates their significant performance impact on scheduling in terms of the incurred task pending delays. The second chapter describes an efficient dynamic resource management framework, called *MROrchestrator*, which alleviates the downsides of slot-based resource allocation in Hadoop MapReduce clusters. *MROrchestrator* monitors and analyzes the execution-time resource footprints of constituent map and reduce tasks, and constructs run-time performance models of tasks as a function of their resource allocations, thereby improving the performance of applications and boosting the cluster resource utilization. The third chapter proposes *HybridMR*, a hierarchical MapReduce scheduler for hybrid data centers. *HybridMR* operates in a 2-phase hierarchical fashion, where the first phase guides placement of MapReduce jobs on native or virtual machines, based on the expected virtualization overheads. The second phase manages the run-time resource allocations of interactive applications and collocated batch MapReduce jobs, with an objective to provide the best effort delivery to the MapReduce jobs, while complying with the Service Level Agreements (SLAs) of the interactive services. Finally, the fourth chapter addresses the reliability of clouds in the context of efficient problem determination and diagnosis in virtualized cloud platforms, through a novel fault management framework, called *CloudPD*. A 3-level hierarchical architecture is adopted by *CloudPD*, consisting of a light-weight event generation phase, comprehensive problem determination phase, and an expert knowledge driven problem diagnosis phase, to provide accurate and fast localization of various anomalies in clouds.

Overall, the dissertation upholds the fact that performance and reliability issues in cloud computing environment are very critical aspects, and need to be well tackled through effective novel research and methodological evaluation.

Table of Contents

List of Figures

List of Tables

Acknowledgments

For the accomplishment of any self defined goal or milestone, motivation and inspiration form an important prop. I owe my inspiration to a group of individuals who with their strong support and championing have made this dissertation an actual realization. The following is my sincere attempt to acknowledge the valuable contributions of each across different contexts.

At first, I would like to express my sincere gratitude towards my dissertation advisor, Chita R. Das. He has been a perpetual source of immense support, guidance, mentorship and motivation all through my graduate period. He has provided me with all the freedom, opportunities and responsibilities, that were quintessential to realize the full potentials of my academic and research acumen, and always steered my visions and goals towards the right direction. His research ethics, professionalism and overall personality have always been an inspiration. I have borrowed from him the qualities of a good researcher, tips to maintain calmness in adverse situations, prioritization, and the spirit of comradeship. All these have helped me immensely throughout the journey and getting to this point, where I am able to write my earnest acknowledgment for him.

I would like to thank professors Mahmut T. Kandemir and Bhuvan Urgaonkar for providing me with their valuable mentorship, and also serving as my dissertation committee members. I thank professor Qian Wang for serving as my dissertation external member.

My sincere thanks to Dr. Joseph L. Hellerstein for agreeing to serve as a special member in my dissertation committee. More than that, I have been very fortunate to have him as my mentor during my summer and fall internships at Google. He is instrumental in having taught me numerous attributes, ranging from excellent software engineering skills, industrial research prowess, good oratory and articulation aptitude, and principles of professionalism. I am highly impressed with his wide range of expertise, and the lessons learned from him will always help me make an intelligent call in decisive life situations.

I have also been very fortunate to get to work with the best lab colleagues and friends in my department. My graduate years would not have been so comfortable and cozy without their comradeship. Their obliging presence has always been felt during my conference submission deadlines, as course project partners, and in various other academic and recreational activities. I will miss their company during lunch, evening snacks and coffee hangouts, where we shared myriad of interesting discussions, ranging from sports, politics, social networks, career, and research ideas. In this context, I would like to extend my special appreciations for my lab mates, Reetuparna Das, Asit Mishra, Seung-Hwan Lim, Gunwoo Nam, Pushkar Patankar, Adwait Jog, Nachiappan Chidambaram, Mahshid Sedghi and Onur Kayiran, Amin Jadidi and Tuba Kesten.

My research and graduate studies have been funded by generous grants from NSF and Google, to which, I am very grateful. My special thanks to Google Inc. and IBM Research India, for giving me the unique and coveted internship opportunities. The industry experience was very valuable in imparting the real-life practical perspectives to my research study. The learning and overall exposure helped me gain confidence and appreciate the correlation between academic research and production environment.

Last but not the least, I lack words to describe the in-depth contributions of my family, especially my father, Bajarang Lal Sharma; my mother, Draupadi Sharma; my wife, Anushree Sharma; and my brother and sisters, in providing me with the ever required emotional support, encouragements and moral boost.

Dedication

Dedicated to my loving parents, wife, brother, sisters and friends

for being a constant source of inspiration, motivation and role models in my life.

Chapter 1

Introduction

Cloud computing has emerged as one of the pivotal future compute platforms [1, 2]. It offers a unique and versatile form of *utility computing*, where both infrastructure and application resources can be leased and delivered as services to end users. Cloud infrastructures differ from traditional cluster systems like grids and data centers, both in terms of the underlying platform and the features provided. The important differentiating characteristics include *elasticity*, which is the ability to scale up and down the resource on demand; *multi-tenancy and shared resource pooling*, that allows numerous applications from various users to co-exist and share the same set of cloud resources in the most beneficial manner; *virtualized consolidated environment*, that enables multiple virtual machines (VMs) to be accommodated on a smaller set of physical servers to mitigate server sprawl [3]. Virtualization is the key enabling technology that leverages its unique features like VM live migration and dynamic VM resizing to provide a cloud platform the abilities to deliver its promises; and *self-organizing*, where a cloud is equipped with all necessary intelligence and capabilities for automatically managing its platform to deal with events like failures and consistency. There are various dimensions of cloud computing, that relate to the different challenges associated with this pay-as-you-go computing paradigm. This includes issues like unpredictable performance, data confidentiality, availability and management of large systems, and many more [4]. This dissertation makes a sincere attempt to address one such facet – *performance and reliability* implications in clouds. The dissertation pursues this in four separate but inter-related perspectives. These are *workload characterization*, *resource management*, *workload scheduling* and *problem determination* in cloud platforms. The specific problem statements and research contributions of this dissertation are described below:

1.1 Dissertation Statement

Improving and optimizing the performance and reliability of cloud platforms through effective workload characterization, resource management, workload scheduling, and fault diagnosis.

1.2 Dissertation Contributions

The first contribution of this dissertation is related to the proper understanding of the workloads that drive cloud systems [5]. Assessment and evaluation of various functionalities of large data centers and clouds requires benchmarks with representative workloads to gauge the performance impact of system changes, assess changes in application codes, machine configurations, and scheduling algorithms. To meet this objective, it is quintessential to construct workload characterizations from which realistic and symbolic performance benchmarks can be derived. Performance insights obtained from thorough understanding of workloads are essential to effectively manage a system. There are different dimensions of cloud workloads that relate to job arrival rate, run-time resource usage and finish time of jobs, each of which in turn manifests into the observed performance impacts. These features of workloads for Google cloud compute clusters have been recently studied in [6, 7, 8]. Besides these, there exists an important workload property, called *task placement constraint*, that tends to have significant impact on performance, as quantified by the task scheduling delay and resource utilization. We analyze the performance impacts of task placement constraints in large compute clusters like those in Google cloud back-end systems [6], and suggest techniques to incorporate them into existing performance benchmarks to make them more realistic and cloud representative. Besides Google infrastructure, constraints also occur predominantly in other compute clusters in the form of scheduling artifacts. Examples include: the Condor system [9] that uses the ClassAds mechanism [10], IBM's load balancer [11], Utopia system [12], and grid toolkits [13]. The main contributions of this research are: (i) demonstrate that task placement constraints impact scheduling performance by incurring a 2 to 6 factor increase in task pending delays, which implies significant tens of minutes of additional wait times; (ii) construct a simple and intuitive model, *Utilization Multiplier*, to quantify the impact of individual constraint on scheduling; and (iii) develop algorithms to synthesize representative task constraints and machine properties, and incorporate this synthesis into existing performance benchmarks towards realistic cloud workloads representation. Chapter 3 explains this portion in detail.

The second contribution of this dissertation is on developing an efficient resource management technique for big data cloud clusters, running representative large scale data processing applications like Google MapReduce [14]. Cloud computing has become analogous to large scale data intensive computing, where huge amount of data needs to be processed and analyzed within the constraints of efficiency and economy of scale. Hadoop cloud computing platform [15] is a leading provider of such massive data analytic services. Hadoop MapReduce [16] is the open source implementation of Google MapReduce, which provides large-scale distributed processing of data across thousands of commodity-based servers. In typical Hadoop MapReduce clusters, applications from multiple users share the same set of resources, and efficient management of these resources is an important design choice with respect to both application performance and cluster resource utilization standpoints. The key challenges in such shared Hadoop MapReduce clusters include the need to automatically manage and optimally control the allocation of resources to multiple applications. Currently, in Hadoop, resources are managed at the granularity of slots. A slot represents a fixed chunk of machine, consisting of a static quantity of CPU, memory and disk. A slot has three main disadvantages, which are related to its fixed-size, static and coarse grained definition.

Consequently, a slot does not provide explicit isolation and global coordination during resource allocation. These deficiencies manifest into poor performance like high latency, low throughput and reduced resource utilization. This observation is backed by practical evidences. For example, a recent analysis on a 2000-node Hadoop cluster at Facebook [17] has shown both the under and over utilization of resources due to significant disparity between task resource demands and assigned resources in MapReduce clusters. Furthermore, similar analysis from a Microsoft production MapReduce cluster indicates that contention for dynamic machine resources like CPU and memory among tasks contributes significantly towards the prominence of stragglers [18]. Towards this pursuit, this research focuses on the efficient management of resources in Hadoop MapReduce clusters. It presents the design and implementation of an efficient resource management framework, *MROrchestrator*, that understands the dynamic resource foot-prints of jobs, and constructs run-time statistical models of task performance as a function of its resource usage/allocation. *MROrchestrator* achieves around 38% reduction in job completion times, and around 25% increase in cluster resource utilization. More details of the foregoing are covered in Chapter 4.

The third contribution of this dissertation is towards efficient scheduling of workload mix (consisting of transactional applications like web server and batch jobs like MapReduce) on a hybrid compute infrastructure. Today's data centers offer two different modes of computing platforms - native clusters and virtual clusters. Both these environments have their own strengths and weaknesses. For example, a native cluster is better for batch workloads like MapReduce from the performance standpoint, but usually suffers from poor utilization, and high hardware and power cost. A virtual cluster, on the other hand, is attractive for transactional workloads from consolidation and cost standpoints, but may not provide competitive performance like a native cluster. Intuitively, a hybrid compute platform consisting of a virtualized as well as a native cluster, should be able to exploit the benefits of both the environments for providing a better cost-effective infrastructure. In this research proposition, we explore this design alternative, which we call *hybrid data center*, and demonstrate its advantages for supporting both interactive and batch workloads. This dissertation presents the design of a 2-phase hierarchical scheduler, called *HybridMR*, for effective resource management of such hybrid environment. In the first phase, *HybridMR* classifies incoming MapReduce jobs based on the expected virtualization overheads, and uses this information to automatically guide placement between physical and virtual machines. In the second phase, *HybridMR* manages the run-time performance of MapReduce jobs collocated with interactive applications in order to provide best effort delivery to batch jobs, while complying with the SLAs of interactive applications. By consolidating batch jobs with over-provisioned foreground applications, the available unused resources are better used, resulting in improved cluster utilization and energy efficiency. Evaluations on a hybrid data center consisting of 24 physical servers and 48 virtual machines, with diverse workload mix of interactive and batch MapReduce applications demonstrate that *HybridMR* can achieve up to 40% improvement in the completion times of MapReduce jobs, while complying with the target SLAs of interactive applications. Compared to native clusters, at the cost of minimal performance penalty, it boosts the resource utilization by 45%, resulting in around 43% energy savings. These results indicate that a hybrid cluster with an efficient scheduling mechanism can provide a cost-effective solution for hosting both interactive and batch workloads. Chapter 5 contains specific details.

The fourth contribution of this dissertation is towards addressing the reliability of cloud systems, which is specifically related to the efficient problem determination and diagnosis in virtualized cloud environments. In clouds, there is a growing trend of increasing number of performance anomalies [19, 20]. A recent survey [21] demonstrates increasing customer reluctance to move to clouds due to poor performance, which can be attributed to various reasons like resource interference, application bugs, and hardware failures. For instance, in this study, it is observed that if the response time of a cloud-hosted web page increases from 4 to 6 seconds, about 33% of the customers will abandon the service and shift to a different cloud provider. Clouds present an automated and dynamic model, which conflicts with the manual/semi-automatic process of problem determination. The applications running inside the cloud often appear opaque to the cloud provider, which makes it non-trivial to get access to fine-grained system and application measurements for problem detection and diagnosis. Compared to traditional distributed systems like cluster grids and data centers, clouds present additional and non-trivial challenges, which can be attributed to the environment that is much more dynamic, large-scale, experiences high frequency of faults, operates autonomically on a shared resource infrastructure. Towards this, the dissertation addresses the issues of problem determination in a multi-tenant dynamic Infrastructure as a Service (IaaS) cloud model. It presents the design and implementation of *CloudPD*, a fault management framework, which addresses the challenges identified with a problem determination framework for clouds, and is geared towards faults that arise due to the various virtualization related cloud activities. *CloudPD* introduces three novel ideas and combines them with known techniques to design an effective methodology for problem determination. The first idea attacks the problem of a non-stationary context by introducing operating context of an application in its resource model. The second idea is using host metrics as a canonical representation of the operating context drastically reducing the number of resource models to be learned. Moreover, the use of an online learning approach further reduces the number of resource models learned by the system. The third idea is a three-level framework (i.e., a light-weight event generation stage, an inexpensive problem determination stage and a robust diagnosis stage) which combines resource models with correlation models as an invariant of application behavior. Using a prototype implementation with cloud representative workloads like Hadoop, Olio and RUBiS, we demonstrate that *CloudPD* detects and diagnoses faults with low false positives and high accuracy of 88%, 83% and 83%, respectively. In an enterprise trace-based case study, *CloudPD* diagnosed problems within 30 seconds with an accuracy of 77%, demonstrating its effectiveness in real-life operations. *CloudPD* is the first end-to-end fault management system that can detect, diagnose, classify and suggest remediation actions for virtualized cloud-based anomalies. Chapter 6 provides more details in this context.

1.3 Dissertation Organization

This dissertation is organized into seven chapters. Chapter 2 contains the overall background of the dissertation main topic, and summary of related research.

Chapter 3 describes the work done on cloud workload characterization in the context of modeling and synthesizing task placement constraints. Chapter 4 presents *MROrchestrator* framework towards effective management of resources in Hadoop MapReduce clusters. Chapter 5 covers the design and implementation of *HybridMR*, a hierarchical MapReduce scheduler for hybrid data centers. Chapter 6 describes *CloudPD*, a problem detection and diagnosis framework for shared virtualized dynamic clouds. The overall summary of the dissertation with pointers to future research directions are outlined in Chapter 7.

Chapter 2

Background

2.1 Background

This chapter starts with describing some of the preliminaries of the overall topic, including a brief primer on cloud computing, virtualization and MapReduce. It then summarizes the prior works in the context of the main contents presented in this dissertation.

2.1.1 Cloud Computing: A Brief Primer

Cloud computing refers to both applications delivered as services over the Internet, as well as the hardware and systems software in the data centers that provide these services [4]. It is the most popular practical realization of computing as a utility, where the hardware and software are available to be leased in a pay-as-you-go manner to the end users. When this utility computing service is made available to the general public, the model is referred as a *Public Cloud*. A *Private Cloud* is used to refer to services housed in internal data centers of private organization, and not accessible to general users. A *Hybrid Cloud* is a composition of public and private cloud models. Some of the unique characteristics of clouds that make them stand out from traditional distributed systems like grids and data centers include: (i) *Elasticity* – on-demand provisioning in terms of procurement and release of resources from an infinite pool of shared computing resources; (ii) *Multi-tenancy* – clouds provide a secured and coordinated shared environment where multiple applications from various users access a common set of hardware and software amenities; (iii) *Self-organizing* – clouds are equipped with sufficient intelligence and automation required to self manage their software and infrastructure premises in a full to semi-autonomic manner; (iv) *Dynamism* – clouds represent an environment which is much larger in scale and dynamic in terms of applications, tenants and facilities. This introduces new challenges in the performance and reliability management of these systems; (v) *Resource Pooling* – clouds providing the illusion of infinite computing resources available on demand to the end users. This precludes the need to plan ahead for provisioning; and (vi) *Cloud Bursting* – it is the process of off-loading tasks to the cloud during times when the compute resources are needed, for example, during flash crowds. The potential capital savings through *cloud bursting* is significant because the organizations are not required to invest in the procurement of additional servers to address peak loads, which occur very rarely. Due to these features and many more, the popularity and usage of cloud computing is growing every day.

The cloud computing stack offers three popular computing models [22]. They are (i) *Infrastructure as a Service (IaaS)* – this model offers cloud resources to be leased in the form of independent raw virtual machines. Cloud users have the responsibility and flexibility to install the operating system and other software stack as required. Examples of IaaS include Amazon EC2 [1] and Google Compute Engine [23]; (ii) *Platform as a Service (PaaS)* – in this model, cloud providers deliver a computing platform typically including operating system, programming language execution environment, web server and database. This allows developers to run their applications without having to invest their time and associated complexity in managing the underlying hardware and software layers. Examples of PaaS include Windows Azure and Force.com; (iii) *Software as a Service (SaaS)* – here, cloud providers install and operate application software, and the users access the software from their end clients. Examples of SaaS include Google Apps and Microsoft Office 365. Recently, there have been other cloud models like *Network as a Service (NaaS)* and *Data as a Service (DaaS)*, which offer network and data focused services, respectively.

Besides the numerous promises and benefits provided by cloud computing, there comes numerous obstacles in realizing this model in its full bloom. Armburst et al. [4] summarized ten prominent obstacles that cloud computing faces, including unpredictability in performance, availability and failures in large systems, data confidentiality, data transfer bottlenecks, and many more. Thus, there is a growing thrust in academic research as well as industry to explore and discover novel and practical methodologies/solutions towards improving this paradigm.

2.1.2 Virtualization

Virtualization has emerged as a key enabling technology for cloud infrastructures. It enables multiple virtual machines (VMs) to be consolidated on a single physical machine (PM), thereby accommodating their workloads onto a subset of available servers. The resource allocation for VMs can be dynamically adjusted and VMs can be provisioned or removed on the fly to cope with dynamic workloads. Virtualization technology provides numerous benefits like server sprawl reduction through workload consolidation, increased resource utilization, and higher energy savings for enterprise data centers and utility clouds. Virtualization through its unique features like live migration, dynamic resizing, reconfiguration and VM snapshotting, ease the management of the underlying infrastructure. Consequently, most cloud platforms like Amazon EC2 [1], Microsoft Azure [24] and RackSpace [25] utilize server virtualization to efficiently share resources among customers, and allow for rapid on-demand elasticity. Virtualization besides its umpteen benefits, also introduces new challenges. For example, compared to traditional distributed systems, problem determination and diagnosis in virtualized cloud environment is more difficult and non-trivial due to the unique features of virtualized clouds like abstracted resources, high workload dynamism and increased scale of monitoring data from multiple VMs consolidated on a subset of servers.

This dissertation focuses on three specific aspects of improving the virtualization related experience in clouds. First, we have designed and implemented *MROrchestrator* for effective management of resources in Hadoop MapReduce clusters running on virtualized platform. Second, we have developed a hierarchical scheduler, called *HybridMR*, for efficient scheduling of workload mix consisting of interactive and batch applications on a hybrid compute infrastructure consisting of both native and virtual machines. Third, our framework, *CloudPD* provides detection, determination and diagnosis of faults or performance anomalies that occur in virtualized cloud platforms.

2.1.3 MapReduce

MapReduce [14] is a parallel programming model for expressing distributed computations on large scale of data, and an execution framework for massive data processing [26]. It was initially pioneered by Google based on fundamental distributed programming paradigms. Hadoop, an initiative from Yahoo!, is the open-source implementation of MapReduce. This paradigm is tightly coupled with *Big Data* [27] phenomenon.

Several academic and commercial organizations use Apache Hadoop [15], an open source implementation of MapReduce ecosystem. In cloud computing environments like Amazon Web Services [28], Hadoop is gaining prominence with services such as Amazon Elastic MapReduce [29], for providing the required backbone for Internet-scale data analytics.

Figure 2.1 shows a generic Hadoop framework. It consists of two main components – a MapReduce engine and a Hadoop Distributed File System (HDFS). It adopts a master/slave architecture, where a single master node runs the software daemons, *JobTracker* (MapReduce master) and *Namenode* (HDFS master), and multiple slave nodes run *TaskTracker* (MapReduce slave) and *Datanode* (HDFS slave). In a typical MapReduce job, the framework divides the input data into multiple splits, which are processed in parallel by map tasks. The output of each map task is stored on the corresponding *TaskTracker's* local disk. This is followed by a shuffle step, in which the intermediate map output is copied across the network, followed by a sort step, and finally the reduce step. In this

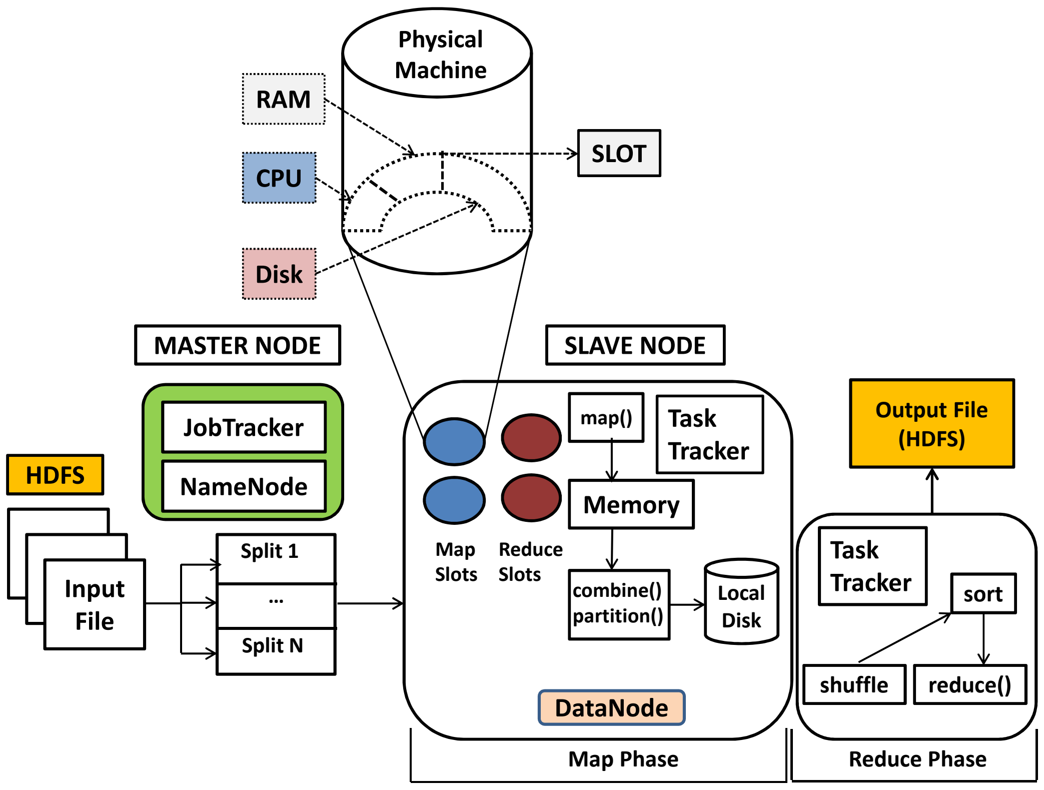


Figure 2.1. Hadoop Framework.

framework, the resources are allocated in the granularity of *slots*, where each slot represents a fixed chunk of a machine's static resource capacity in terms of CPU, memory and disk. The top portion of Figure 2.1 shows the conceptual view of a slot. Only one map/reduce task can run per slot at a time. The primary advantage of a slot is its simplicity and ease of implementation of the MapReduce paradigm. A slot offers a simple but coarse abstraction of the available static resources on a machine. It provides a means to cap the maximum degree of parallelization. However, static, coarse-grained and uniform definition of slots also leads to resource inefficiency and poor performance.

2.2 Summary of Prior Work

This section summarizes the state-of-art in prior works in areas related to the focus of this dissertation. This is categorized into the following sub-sections:

2.2.1 Related Research on Workload Characterization

Existing works relevant to ours which are related to characterization and synthetic generation of workloads, can be broken down into the following sub-categories:

Workload Characterization: Workload characterization is an important artifact for building and maintaining large distributed systems like data centers and clouds. Prominent web server workload characterization works include [30, 31, 32]. The characterization and modeling of scientific and high performance computing workloads are studied in [33, 34, 35, 36]. Workload characterization from other perspectives including large scale distributed file systems, DRAM errors and network traffic, are addressed in [37, 38, 39]. Generic techniques for workload modeling and performance evaluations of computer systems are summarized in [40, 41, 42, 43, 44]. The workload analysis of large scale distributed file systems is discussed in [39]. The characterization of disk drive workloads measured in systems representing the enterprise, desktop, and consumer electronics environments is summarized in [45]. Modeling of High Performance Computing (HPC) workloads has been studied in [33, 34]. Mishra et al. [6] characterized Google workload with focus on resource consumption of tasks running in Google data centers using statistical clustering. Chen et al. [46] summarized the statistical characteristics of the publicly released Google cluster data [47]. There are several contemporary studies on MapReduce [14] workload characterizations [48, 49, 50]. Most of the above works on workload characterization have focused on aspects of workloads related to the resource consumption, arrival patterns, job size, network traffic and file size distribution, with little regard to an important workload property called *task placement constraint*.

Workload Generator and Benchmarking: Workload characterization is fundamental to the synthesis of realistic workloads and the resulting characterizations are used in designing realistic benchmarking tools. Synthetic generation of realistic workloads is important to benchmark and evaluate the performance of a system under study. Many workload drivers exist in literature that generate workloads representative of real applications. Kao *et al.* [51]

proposed a user-oriented synthetic workload generator that simulates file access behaviors of users. WGCap [52] is a synthetic workload generator that simulates the consumption of resources like CPU, memory, disk and network bandwidth for a virtualized capacity planning tool. Shivam et al. proposed an automated framework for storage server benchmarking. GISMO [53] is a workload generator for streaming Internet media objects. Li et al. [54] describe a synthetic workload generator for scientific literature digital libraries and search engines. MapReduce workload generation tools based on statistical and empirical models are discussed in [55, 56, 49]. Popular Web 1.0 benchmarking tools such as *ab*, *httpperf*, SURGE and applications like RUBiS, TPC-W are representative of synthetic Web 1.0 workloads. Recently, a Web 2.0 benchmarking tool called Cloudstone [57] was proposed. Yahoo! Cloud Servicing Benchmark [58] compares the performance of different cloud services using a representative set of workloads. Benchmark suites like Gridmix [59] and HiBench [60] are used for the performance characterization and evaluation of Hadoop framework. MRBench [61] is a benchmark for evaluating the performance of MapReduce systems. We believe the integration of logical constraints related to tasks and machines will help make these benchmarking tools more more realistic and robust in terms of evaluating the performance of the system under test.

Workload Management System: There are existing workload management systems that encapsulate the notion of logical constraints for scheduling tasks across machines in server farms. The Condor project [9] is a popular workload management system that uses logical constraints [62, 10] for resource allocations. IBM's commercial load balancer [11] is based on Condor [9]. Other distributed resource allocation systems which leverage constraints in tasks scheduling include [13, 63, 12].

To the best of our knowledge, we are not aware of any in-depth study of logical constraints with respect to (a) quantification of the performance impact of logical constraints; (b) characterizations of logical constraints; (c) techniques to integrate logical constraints into existing workloads to create realistic benchmarks, representative of a large distributed system like Google. One of the main contributions of this dissertation is to address these missing pieces.

2.2.2 Related Research on Resource Management in Hadoop MapReduce Clusters

In the context of Hadoop, techniques for dynamic resource allocation and isolation have been recently addressed. Polt et al. [64] proposed a task scheduler for Hadoop that performs dynamic resource adjustments to jobs, based on their estimated completion times. Qin et al. [65] leveraged kernel-level virtualization techniques to reduce the resource contention among concurrent MapReduce jobs. Technique for assigning jobs to separate job-queues based on their resource demands was discussed in [66]. ARIA [67] resource manager for Hadoop estimates the amount of resources in terms of the number of map and reduce slots required to meet a given Service Level Agreement (SLA). Another category of work has proposed different resource scheduling policies for Hadoop [68, 69, 70, 71]. The main focus of these schemes is to assign equal resource shares to jobs to maximize resource utilization and system throughput. There are popular resource scheduling managers that manage frameworks like Hadoop. Mesos [72] is a resource scheduling manager that provides fair share of resources across diverse cluster computing frameworks like Hadoop and MPI. Ghodsi et al. [17] proposed a Dominant Resource Fairness (DRF) scheduling algorithm to provide fair allocation of slots to jobs, and is implemented in Mesos. Next Generation MapReduce (NGM) [73] is the most recently proposed architecture of Hadoop MapReduce. It includes a generic resource model for efficient scheduling of cluster resources. NGM replaces the default fixed-size slot with an another basic unit of resource allocation called resource container. However, all the above solutions do not address the fundamental cause of performance bottlenecks in Hadoop, which is related to the static and fixed-size slot-level resource allocation. The current Hadoop schedulers [68, 69] are also oblivious of the run-time resource profiles and demands of tasks.

2.2.3 Related Research on Resource Scheduling in Hadoop MapReduce Clusters

Scheduling techniques for dynamic resource management of MapReduce jobs have been recently addressed. Most of the works in this category primarily focus on different scheduling policies for MapReduce jobs to reduce their completion times, improve cluster resource utilization, energy efficiency and fairness. The default scheduling algorithm in Hadoop is based on the FIFO order, where jobs are executed in the order of their submission. The Fair scheduler [69] targets to give every user a fair share of the cluster capacity. The Capacity scheduler [68] aims to ensure a fair allocation of computation resources amongst users. There are also further optimizations of these schedulers. For example, LATE scheduling algorithm [74] improves upon the speculative executions of straggler tasks in a heterogeneous environment. Delay scheduling [75] overcomes the head-of-line scheduling problem and sticky slots based locality issues. Sandholm et al. [71] proposed Dynamic Priority scheduler to support dynamic capacity distribution among concurrent users based on their priorities. Deadline Constraint scheduler [76] leverages deadline-based scheduling approach to improve cluster resource utilization.

We believe our current work is complementary to these systems in that we share the same motivations and end goals, but we attempt to provide a different approach to handle the same problem, with a coordinated, fine-grained and dynamic resource management framework, called *MROrchestrator*.

2.2.4 Related Research on MapReduce and Virtualization

There has been a recent push to improve the performance of MapReduce on virtualized platforms. Amazon Elastic MapReduce [29] is a publicly offered web service that runs on virtualized cloud platform. Serengeti [77] is an open source project initiated by VMware to enable rapid deployment of Hadoop cluster on a virtual platform. Cardosa et al. [78] proposed techniques for MapReduce provisioning in virtual cloud clusters, with emphasis on energy reduction.

Managing MapReduce workloads using Amazon virtual spot instances is studied in [79]. Resource allocation in Hadoop MapReduce cluster using dynamic prioritization is proposed in [80]. Preliminary evaluations of Hadoop MapReduce performance on virtualized clusters is recently done in [81, 82]. Virtual machine scheduling heuristic to improve Xen scheduler, targeting MapReduce workloads, is addressed in [83]. Harnessing unused CPU cycles in interactive clouds for batch MapReduce workloads has recently been motivated in [84]. Bu et al. [85] proposed an interference-aware and locality-aware scheduling algorithm for optimizing MapReduce in virtualized environment.

2.2.5 Related Research on Interference-based Resource Management in Clouds

With the advent of cloud computing, resource management in virtualized clouds has emerged as an important research avenue. There exists quite a few literatures in this context. For example, Q-Clouds [86] leverages online feedback control technique to dynamically manage the resource allocation to VMs. TRACON [87] is an interference-aware scheduling algorithm for data-intensive applications in virtual environment. Automatic resource provisioning in MapReduce cloud clusters is explored in [78]. Koh et al. [88] presents an empirical study on the performance interference effects in virtual environments. Pu et al. [89] is an experimental research on performance interference in parallel processing of CPU-intensive and network-intensive workloads on Xen virtual machine monitor.

We share our motivation of hybrid cloud clusters with [90, 84, 91]. We believe our work differs from others in the following manner. First, a detailed empirical evaluation and performance analysis study of Hadoop MapReduce on virtual environment have not been addressed in prior literature. Second, scheduling of heterogeneous workloads (mix of transactional and batch MapReduce jobs) on a hybrid cluster (consisting of both native and virtual environments) to exploit the spare resources available due to over-provisioning of interactive applications, has not been explored before. Third, our proposed hierarchical scheduler, *HybridMR*, uniquely focuses on the performance enhancement of MapReduce jobs, collocated with other interactive jobs in a virtual environment while complying with the SLAs of the foreground jobs. Fourth, no previous studies have paid much attention to the benefits and design trade-offs of hybrid compute clusters consisting of both native and virtual servers, hosting heterogeneous workloads.

2.2.6 Related Research on Fault Diagnosis in Clouds

Problem determination in distributed systems in general is an important system management artifact and has been thoroughly studied in prior literature. Some of these studies can be classified into the following three categories based on the type of techniques used, as described below:
(a) *Threshold-based schemes:* In this approach, an upper/lower bound is set for each system metric being monitored. These thresholds are determined based on the historical data analysis or predefined application-level performance constraints like QoS or SLAs. On violation of the threshold for any metric being monitored, an anomaly alarm is triggered. This methodology forms the core of many of the commercial [92, 93] and some open source [94, 95] monitoring tools. This methodology however suffers from high false alarm rate, static and off-line characteristics, and is expected to perform poorly in the context of large scale utility clouds [96].
(b) *Statistical machine learning techniques:* Many anomaly detection schemes leverage machine learning and statistical methods. Anomaly detection tools like like E2EProf [97] and Pinpoint [98] use various statistical techniques like correlation, clustering, entropy, profiling and analytical modeling, for the identification of performance related problems in distributed systems.

(c) *Problem determination in clouds:* Recently, researchers have started to address fault management in virtualized systems. Kang et al. [99] proposed PeerWatch, a fault detection and diagnosis tool for virtualized consolidated systems. PeerWatch utilizes a statistical technique, cannonical correlation analysis to model the correlation between multiple application instances to detect and localize faults. EbAT [96] is a system for anomaly identification in data centers. EbAT analyzes system metric distributions rather than individual metric thresholds. Vigilant [100] is an out-of-band, hypervisor-based failure monitoring scheme for virtual machines, that uses machine learning to identify faults in the VMs and guest OS. Yehuda et al. [101] presented an application-agnostic approach for multi-tiered systems hosted on virtualized environments. Log-based troubleshooting system for cloud infrastructures is described in [102]. Cherkasova et al. [103] focused on differentiating anomaly from application change and workload change. Bare et al. [104] proposed an automated online framework for performance diagnosis in traditional distributed systems. DAPA [105] is an initial prototype of application performance diagnostic framework for a virtualized environment. PREPARE [106] is a recently proposed framework for performance anomaly prevention in virtualized clouds. It integrates online anomaly prediction and predictive prevention to minimize the performance anomaly penalty. The main focus of all these initial works in the context of virtualized environment is to diagnosis application performance anomalies and identify causes of SLA violations. A recent study [20] on 3 years worth forum messages concerning the problems faced by end users of utility clouds show that virtualization related problems contribute to around 20% of the total problems experienced.

The above mentioned prior works both in the context of traditional distributed systems and utility clouds have only addressed software bug or application related faults, which manifest into performance anomalies. However, none of these have focused on cloud-centric faults that arise from frequent, dynamic reconfiguration activities in clouds like wrong VM sizing, VM live migration, and anomalies due to collocation, *i.e.,* anomalies that arise due to sharing of resources across VMs consolidated on the same physical hardware. The other important dimensions where our proposed fault management framework, *CloudPD*, differ from the prior related works include: (i) we monitor and analyze a diverse and complete list of system metrics (see Table 6.1) to better capture the system context, compared to only CPU and memory metrics being considered in most prior works; (ii) most prior literature talk about efficient anomaly detection, but very few go beyond that and address diagnosis of these faults after detection. The Diagnosis Engine of *CloudPD* localizes the root cause of fault in terms of affected system metrics, VM, server, and application component. Further, it also classifies these anomalies into known fault type for better handling of similar future anomalies. Moreover, the Anomaly Remediation Manager handles preventive and remedial actions by coordinating with other cloud modules. These pieces are absent in previous works; (iii) a narrow range of application benchmarks are considered for evaluations in previous works. We evaluate *CloudPD* with three representative cloud workloads – Hadoop, Olio and RUBiS, and also a case study with real traces from an enterprise application; (iv) small experimental test-bed has been used in previous studies (5-8 VMs). We have used a comparatively larger evaluation system consisting of 28 virtual machines, consolidated on 4 blade servers. Thus, to the best of our knowledge, we believe to be the first to address fault/anomaly detection and localization for cloud-centric virtualized infrastructure, and covering a larger set of applications/workloads and experimental test-bed.

Chapter 3

Modeling and Synthesizing Task Placement Constraints in Cloud Clusters

3.1 Introduction

Building compute clusters at Google scale requires having realistic benchmarks to evaluate the performance impact of changes in scheduling algorithms, machine configurations, and application codes. Developing such benchmarks requires constructing workload characterizations that are sufficient to reproduce key performance characteristics of compute clusters. Existing workload characterizations for high performance computing and grids focus on task resource requirements such as CPU, RAM, disk and network. However, in addition to resource requirements, Google tasks frequently have *task placement constraints* (hereafter, just *constraints*) similar to the Condor ClassAds mechanism [62]. Examples of constraints are restrictions on task placement due to hardware architecture and kernel version. Constraints limit the machines on which a task can run, and this in turn can increase task scheduling delays. This chapter develops methodologies that quantify the performance impact of task placement constraints, and applies these methodologies to Google compute clusters. In particular, this chapter develops a methodology for synthesizing task placement constraints and machine properties to provide more realistic performance benchmarks. Herein, task scheduling refers to the assignment of tasks to machines. Delays that occur once a task is assigned to a machine (e.g., delays due to operating system schedulers) are not considered since as observed, these delays are much shorter than the delays for machine assignments.

This chapter elaborates on the difference between task resource requirements and task placement constraints. Task resource requirements describe *how much* resource a task consumes. For example, a task may require 1.2 cores per second, 2.1 GB of RAM per second, and 100 MB of disk space. In contrast, task placement constraints address *which* resources are consumed. A common constraint in Google compute clusters is requiring a particular version of the kernel (e.g., because of task dependencies on particular APIs). This constraint has no impact on the quantities of resource consumed. However, the constraint does affect the machines on which tasks can schedule. The constraints herein addressed are simple predicates on machine properties. Such constraints can be expressed as a triple of: machine attribute, relational operator, and a constant. An example is "kernel version is greater than 1.2.7".

Why do Google tasks specify constraints? One reason is machine heterogeneity. Machine heterogeneity arises because financial and logistical considerations make it almost impossible to have identical machine configurations in large compute clusters. As a result, there can be incompatibilities between the pre-requisites for running an application and the configuration of some machines in the compute cluster (e.g., kernel version). To address these concerns, Google tasks may request specific hardware architectures and kernel versions. A second reason for task placement constraints is application optimization, such as making CPU/memory/disk trade-offs that result in tasks preferring specific machine configurations. For these reasons, Google tasks will often request machine configurations with a minimum number of CPUs or disks. A third reason for task constraints is problem avoidance. For example, administrators might use a clock speed constraint for a task that is observed to have errors less frequently if the task avoids machines that have slow clock speeds.

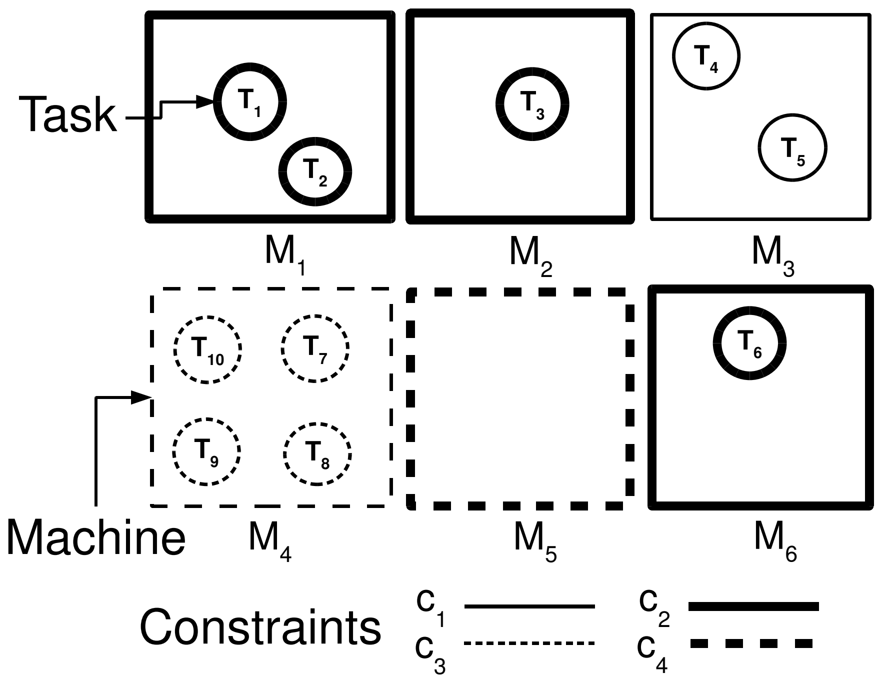


Figure 3.1. Illustration of the impact of constraints on machine utilization in a compute cluster. Constraints are indicated by a combination of line thickness and style. Tasks can schedule only on machines that have the corresponding line thickness and style.

Figure 3.1 illustrates the impact of constraints on machine utilization in a compute cluster. There are six machines $\mathrm{M}_1, \cdots, \mathrm{M}_6$ (depicted by squares) and ten tasks $\mathrm{T}_1, \cdots, \mathrm{T}_{10}$ (depicted by circles). There are four constraints $\mathrm{c}_1, \cdots, \mathrm{c}_4$. Constraints are indicated by the combinations of line thickness and line styles. In this example, each task requests a single constraint, and each machine satisfies a single constraint. A task can only be assigned to a machine that satisfies its constraint; that is, the line style and thickness of a circle must be the same as its containing square. One way to quantify machine utilization is the ratio of tasks to machines. In the example, the average machine utilization is 10 tasks $\div$ 6 machines $=$ 1.66 tasks per machine. However, tasks with constraint c_3 can be scheduled only on machine M_4 where there are 4 tasks. So, the utilization seen by a newly arriving task that requests c_3 is 4 tasks $\div$ 1 machine $=$ 4 tasks per machine. Now consider c_2. There are four tasks that request constraint c_2, and these tasks can run on three machines (M_1, M_2, M_6). So, the average utilization experienced by a newly arriving task that requests c_2 is 4 tasks $\div$ 3 machine $=$ 1.33 tasks per machine. In practice, it is more complicated to compute the effect

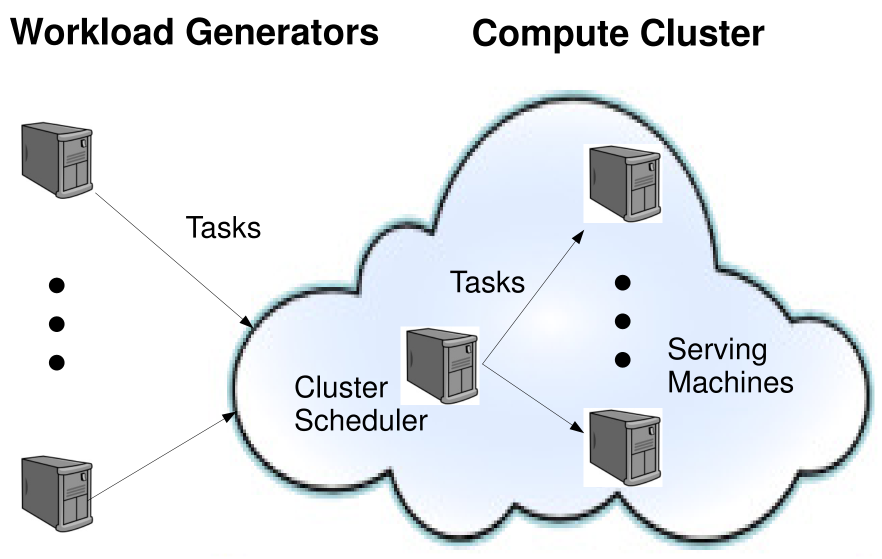


Figure 3.2. Components of a compute cluster performance benchmark.

of constraints on resource utilization because: (a) tasks often request multiple constraints; (b) machines commonly satisfy multiple constraints; and (c) machine utilization is a poor way to quantify the effect of constraints in compute clusters with heterogeneous machine configurations.

This chapter uses two metrics to quantify the performance impact of task placement constraints. The first metric is *task scheduling delay*, the time that a task waits until it is assigned to a machine that satisfies the task constraints. Task scheduling delay is the primary metric by which performance assessments are done in Google compute clusters because most resources are consumed by tasks that run for weeks or months [6]. An example is a long running search task that alternates between waiting for and processing user search terms. A cluster typically schedules 5 to 10 long-running tasks per hour, but there are bursts in which a hundred or more tasks must be scheduled within minutes. For long-running tasks, metrics such as response time and throughput have little meaning. Instead, the concern is minimizing task scheduling delays when tasks are scheduled initially and when running tasks are rescheduled (e.g., due to machine failures). Our second metric is *machine resource utilization*, the fraction of machine resources that are consumed by scheduled tasks. In general, high resource utilization is desired to achieve a better return on the investment in compute clusters.

Much of the chapter's focus is on developing realistic performance benchmarks. As depicted in Figure 3.2, a benchmark has a workload generation component that generates synthetic tasks that are scheduled by the Cluster Scheduler and executed on Serving Machines. Incorporating task placement constraints into a performance benchmark requires changes to: (a) the Workload Generators to synthesize tasks so that they request representative constraints; and (b) the properties of Serving Machines so that they are representative of machines in production compute clusters.

Thus far, the discussion has focused on task placement constraints related to machine properties. However, there are more complex constraints as well. For example, a job may request that no more than two of its tasks run on the same machine (e.g., for fault tolerance). Although, there is a plan to address the full range of constraints in the future, the initial efforts are more modest. Another justification for the chapter's limited scope is that complex constraints are less common in Google workloads. Typically, only 11% of the production jobs use complex constraints. However, approximately 50% of the production jobs have constraints on machine properties.

To the best of our knowledge, this is the first research effort to study the performance impact of task placement constraints. It is also the first endeavor to construct performance benchmarks that incorporate task placement constraints. The specifics of this chapter's contributions are best described as answers to a series of related questions.

Q1: Do task placement constraints have a significant impact on task scheduling delays? We answer this question using benchmarks of Google compute clusters. The results indicate that the presence of constraints increases task scheduling delays by a factor of 2 to 6, which often means tens of minutes of additional task wait time.

Q2: Is there a model of constraints that predicts their impact on task scheduling delays? Such a model can provide a systematic approach to re-engineering tasks to reduce scheduling delays and configuring machines in a cost-effective manner. We argue that task scheduling delays can be explained by extending the concept of resource utilization to include constraints. To this end, we develop a new metric, the **Utilization Multiplier (UM)**. UM is the ratio of the resource utilization seen by tasks with a constraint to the average utilization of the resource. For example, in Figure 3.1, the UM for constraint c_3 is $\frac{4}{1.66} = 2.4$ (assuming that there is a single machine resource, machines have identical configurations, and tasks have identical resource demands). As discussed in Section 3.4, UM provides a simple model of the performance impact of constraints in that task scheduling delays increase with UM.

Q3: How can task placement constraints be incorporated into existing performance benchmarks? This chapter describes how to synthesize representative task constraints and machine properties, and how to incorporate this synthesis into existing performance benchmarks. We find that our approach accurately reproduces performance metrics for benchmarks of Google compute clusters with a discrepancy of only 13% in task scheduling delay and 5% in resource utilization.

The remainder of this chapter is organized as follows: Section 3.2 describes our experimental methodology. Section 3.3 assesses the impact of constraints on task scheduling delays. Section 3.4 constructs a simple model of the impact of constraints on task scheduling delays. Section 3.5 describes how to extend existing performance benchmarks to incorporate constraints. Section 3.6 contains the summary of this chapter.

3.2 Experimental Methodology

This section describes the Google task scheduling mechanism and our experimental methodology.

Our experiments use data from three Google compute clusters. We refer to these clusters as A, B and C. The clusters are typical in that: (a) there is no dominant application; (b) there are thousands of machines; and (c) the cluster runs hundreds of thousands of tasks in a day.

There are four task types. Tasks of type 1 are high priority production tasks; tasks of type 4 are low priority, and are not critical to end-user interactions; tasks of type 2 and 3 have characteristics that blend elements of task types 1 and 4. Figure 3.3 shows the fraction of tasks by type in the Google compute clusters. These fractions are used in Section 3.5 to construct workloads with representative task placement constraints.

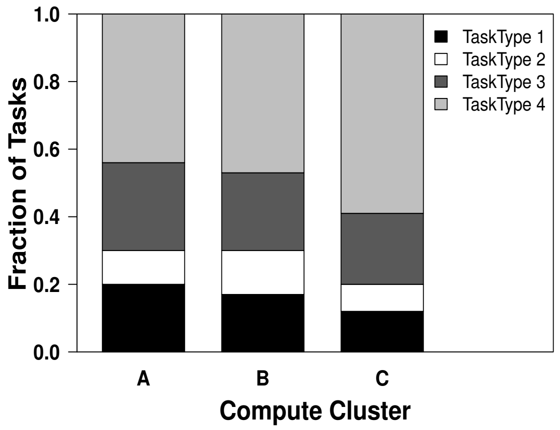


Figure 3.3. Fraction of tasks by type in Google compute clusters.

3.2.1 Google Task Scheduling

Next, we describe how scheduling works in Google compute clusters. Users submit jobs to the Google cluster scheduler. A job describes one or more tasks [6]. The cluster scheduler assigns tasks to machines. A task specifies (possibly implicitly) resource requirements (e.g., CPU, memory, and disk resources). A task may also have task placement constraints (e.g., kernel version).

In principle, scheduling is done in order by task type, and is first-come-first-serve for tasks with the same type. Scheduling a task proceeds as follows:

- Determine which machines satisfy the task's constraints.
- Compute the subset of machines that also have sufficient free resource capacity to satisfy the task's resource requirements (called the *feasible set*).
- Select the *best* machine in the feasible set on which to run the task (assuming that the feasible set is not empty).

To elaborate on the last step, selecting the *best* machine does optimizations such as balancing resource demands across machines and minimizing peak demands within the power distribution infrastructure. Machines notify the scheduler when a job terminates, and machines periodically provide statistics so that the cluster scheduler has current information on machine resource consumption.

3.2.2 Methodology

We now describe our methodology for conducting empirical studies. A study is two or more experiments whose results are compared to investigate the effects of constraints on task scheduling delays and/or machine resource utilization. Figure 3.4 depicts the workflow used in our studies. There are four sub-workflows. The *data preparation sub-workflow* acquires raw trace data from production Google compute clusters. A raw trace is a kind of scheduler checkpoint (e.g., [10]) that contains the history of all scheduling events along with task resource requirements and placement constraints.

The *baseline sub-workflow* runs experiments in which there is no modification to the raw trace. This sub-workflow makes use of benchmarks that have been developed for Google compute clusters. The benchmarks are structured as in Figure 3.2. *Workload Generation* is done by synthesizing tasks from traces of Google compute clusters. The real Google cluster scheduler then makes the scheduling decisions. One version of the benchmark runs with real *Serving Machines*. In a second version of the benchmark, there are no *Serving Machines*; instead, the *Serving Machines* are mocked using trace data to provide statistics of task executions on *Serving Machines*. Our studies use the latter benchmark for two reasons. First, it is unnecessary to use real *Serving Machines*. This is because once task assignments are known, task scheduling delays and machine resource utilization can be accurately estimated from the task execution statistics in the traces. Second, and more importantly, it is cumbersome at best to use real *Serving Machines* in our studies since evaluating the impact of task constraints requires an ability to modify machine properties. The baseline sub-workflow produces *Baseline Benchmark Results*.

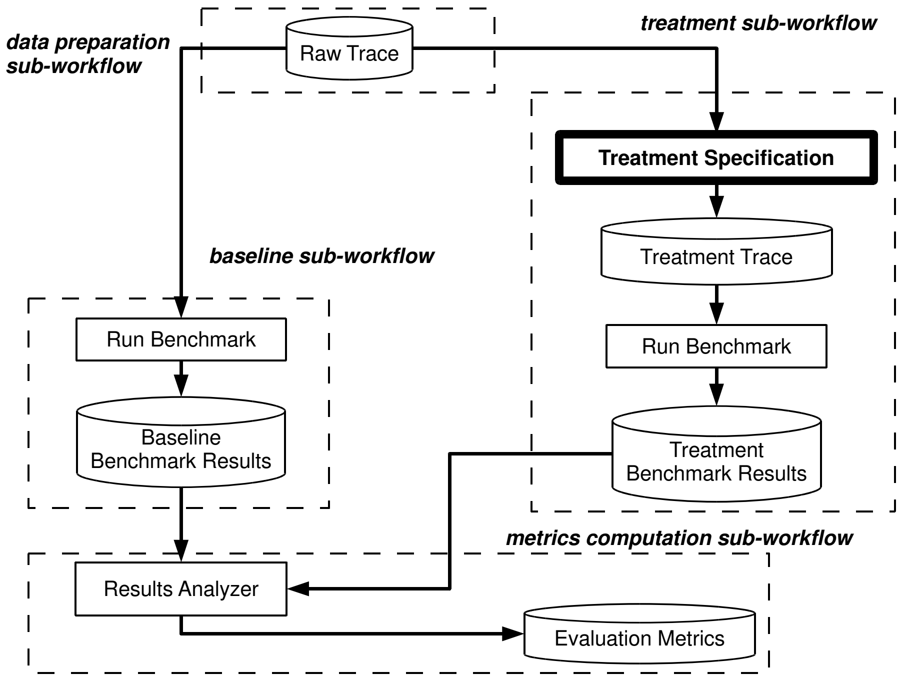


Figure 3.4. Workflow used for empirical studies. The *Treatment Specification* block is customized to perform different benchmark studies.

The *treatment sub-workflow* performs experiments in which machine properties and/or task constraints are modified from those in the raw trace resulting in a Treatment Trace. The block labeled **Treatment Specification** performs the modifications to the raw trace for an experiment. For example, in the next section, the *Treatment Specification* removes all constraints from tasks in the raw trace. This sub-workflow produces *Treatment Benchmark Results.*

These computations use the Results Analyzer, which inputs the *Baseline Benchmark Results* and *Treatment Benchmark Results* to compute evaluation metrics for task scheduling delays and machine resource utilization. Our studies employ raw traces from the above mentioned three Google compute clusters. (Although many tasks are scheduled in a day, most consume few resources.) We use a total of 15 raw traces, with 5 traces from each of the three compute clusters. The raw traces are obtained at the same time on successive days during a work week. Because scheduling considerations are more important when resources are scarce, we select traces that have higher resource utilization.

3.3 Performance Impact of Constraints

This section addresses the question **Q1: Do task placement constraints have a significant impact on task scheduling delays?** Answering this question requires considering two factors in combination: (1) the supply of machine resources that satisfy constraints and (2) the resources demanded by tasks requesting constraints.

The constraints satisfied by a machine are determined by the machine's properties. We express machine properties as attribute-value pairs. Table 3.1 shows machine attributes, the short names of attributes that are used in this chapter, and the number of possible values for each machine attribute. To avoid revealing details of Google machine configurations, we do not list the *values* of the machine attributes.

We use Table 3.1 to infer the number of possible constraints. Recall that a constraint is a triple of machine attribute, relational operator, and value. We only consider constraints that use attribute values of machine properties since constraints that use other values are equivalent to constraints that use values of machine properties. For example, "$num\_cores > 9$" is equivalent to "$num\_cores > 8$" if the maximum value of num_cores is 8. It remains to count the combinations of relational operators and machine properties. For categorical variables, there are two possible relational operators ($\{=, \neq\}$), and for numeric variables there are 6 possible relational operators ($\{=, \neq, \leq, <, \geq, >\}$). Thus, the number of feasible constraints is $\sum_i v_i r_i \approx 400$, where v_i is the number of values of the i-th machine attribute and r_i is the number of relational operators that can be used with the machine attribute.

Not surprisingly, it turns out that only a subset of the possible constraints are used in practice. Table 3.2 lists the thirty-five constraints that are commonly requested by tasks. The constraint type refers to a group of constraints with similar semantics. With two exceptions, the constraint type is the same as the machine attribute. The two exceptions are *max_disks* and *min_disks*, both of which use the *num_disks* machine attribute. For the commonly requested constraints, the

Short Name	Description	# of values
arch	architecture	2
num_cores	number of cores	8
num_disks	number of disks	21
num_cpus	number of CPUs	8
kernel	kernel version	7
clock_speed	CPU clock speed	19
eth_speed	Ethernet speed	7
platform	Platform family	8

Table 3.1. Machine attributes that are commonly used in scheduling decisions. The table displays the number of possible values for each attribute in Google compute clusters.

Constraint Names	Constraint Type	Relational Operator
c1.{1}.{1}	arch	$=$
c2.{1-5}.{1-2}	num_cores	$=, \geq$
c3.{1-3}.{1-2}	max_disks	$=, \geq$
c4.{1-2}.{1-2}	min_disks	$=, \geq$
c5.{1-4}.{1-2}	num_cpus	$=, \geq$
c6.{1-2}.{1}	kernel	$=$
c7.{1-2}.{1}	clock_speed	$=$
c8.{1}.{1}	eth_speed	$=$
c9.{1}.{1}	platform	$=$

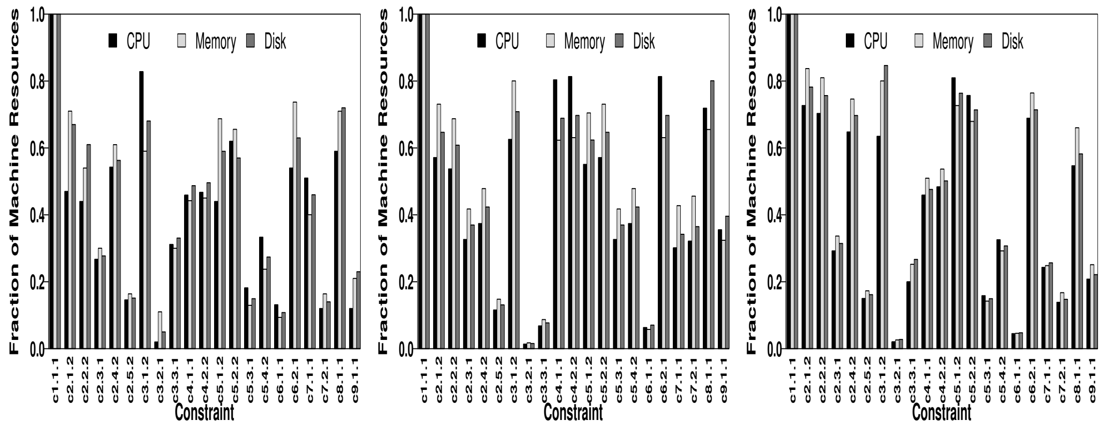
Table 3.2. Popular task constraints in Google compute clusters. The constraint name encodes the machine attribute, property value, and relational operator.

relational operator is either $=$ or $\geq$. Note that $\geq$ is used with *max_disks* and *min_disks*, although the intended semantics is unclear. One explanation is that these are mistakes in job configurations.

The constraint names in Table 3.2 correspond to the structure of constraints. Our notation is as follows:
c <constraint type>.<attribute index>.<relational operator index>. For example, "c2.4.2" is a *num_cores* constraint, and so it begins with "c2" since *num_cores* is the second constraint type listed in Table 3.2. The "4" specifies the *index* of the value of number of cores used in the constraint (but 4 is not necessarily the value of the attribute that is used in the constraint). The final

"2" encodes the $\geq$ relational operator. In general, we encode the relational operators using the indices 1 and 2 to represent $=$ and $\geq$, respectively.

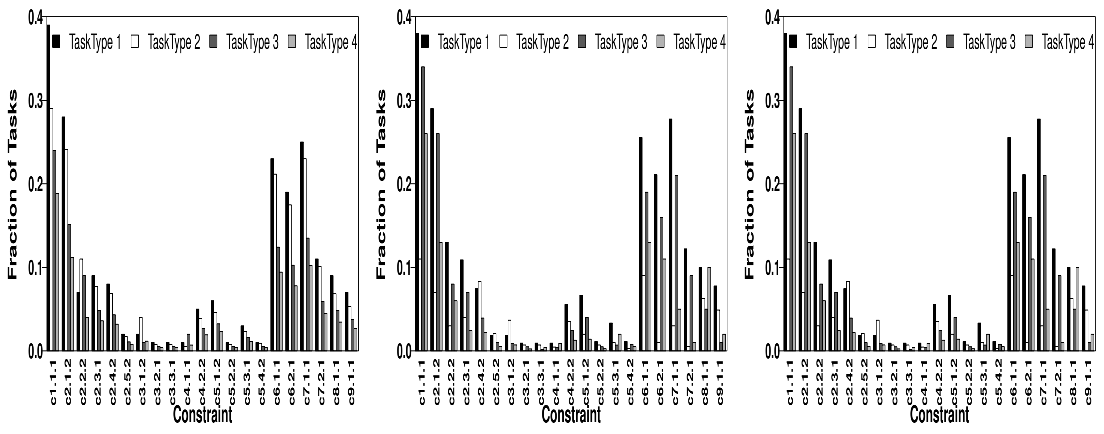
We provide more insights into the constraints in Table 3.2. The *num_cores* constraint requests a number of physical cores, which is often done to ensure sufficient parallelism for application codes. The *max_disks* constraint requests an upper bound on the number of disks on the machine, typically to avoid being collocated with I/O intensive workloads. The *min_disks* constraint requests a minimum number of disks on the machine, a common request for I/O intensive applications. The *kernel* constraint requests a particular kernel version, typically because the application codes depend on certain kernel APIs. The *eth_speed* constraint requests a network interface of a certain bandwidth, an important consideration for network-intensive applications. The remaining constraints are largely used to identify characteristics of the hardware architecture. The constraints included here are: *arch, clock_speed, num_cpus, and platform.*



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 3.5. Fraction of compute cluster resources on machines that satisfy constraints.

We now describe the supply of machine resources that satisfy constraints. Figure 3.5 plots the supply of compute cluster CPU, memory, and disk resources on machines that satisfy constraints. The horizontal axis is the constraint using the naming convention in Table 3.2. Hereafter, we focus on the 21 constraints (of the total 35 constraints shown in Table 3.2) that are most commonly specified by Google tasks. These constraints are the labels of the X-axis of Figure 3.5. The

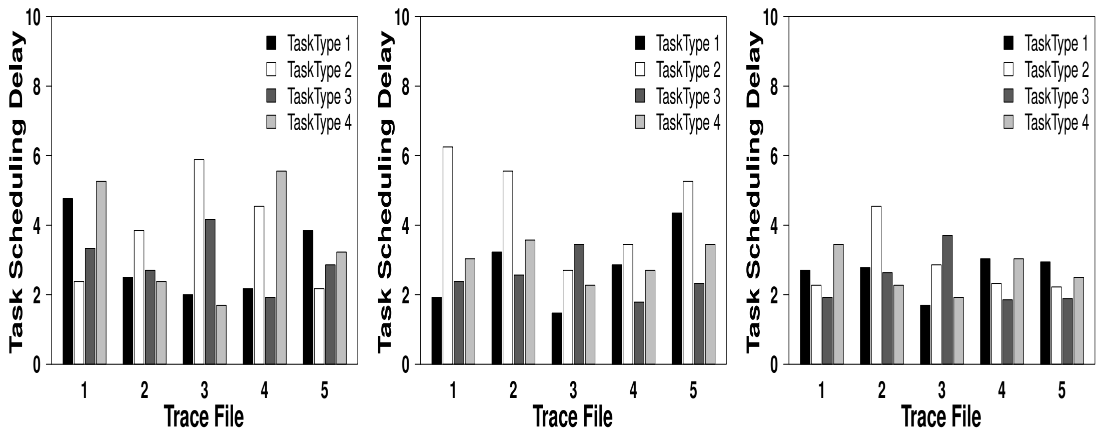


(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 3.6. Fraction of tasks that have individual constraints.

vertical Y-axis of that figure is the fraction of the compute cluster resources that satisfy the constraint, with a separate bar for each resource for each constraint. There is much evidence of machine heterogeneity in these data. For example, constraint c3.1.2 is satisfied by machines accounting for 85% of the CPU of compute cluster A, but these machines account for only 60% of the memory of compute cluster A. On the other hand, constraint c6.2.1 is satisfied by machines that account for only 52% of the CPU of compute cluster A but 75% of the memory.

Next, we consider task demands for constraints. Figure 3.6 demonstrates the demand by task type for compute cluster resources that satisfy the constraints in Table 3.2. These data are organized by task type. The horizontal axis is the constraint, and the vertical axis is the fraction of the tasks (by type) that request the constraint. Note that it is very common for tasks to request the machine architecture constraint (c1.1.1). This seems strange since from Figure 3.5 we see that all machines satisfy c1.1.1 in the compute clusters that we study. One reason may be that historically there has been a diversity of machine architectures. Another possible explanation is that the same task may run in other compute clusters, where constraint c1.1.1 does affect scheduling. Other popular constraints are: the number of cores (c2.*.*), the kernel release (c6.*.*), and the CPU clock speed (c7.*.*).



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 3.7. Normalized task scheduling delay, the ratio of the task scheduling delays with constraints to task scheduling delays when constraints are removed.

We note in passing that even more insights can be provided by extending Figure 3.6 to include the resources requested by tasks. However, these are high dimension data, and so they are challenging to present. Further, these data do not provide a complete picture of the impact of constraints on task scheduling delays since scheduling decisions depend on *all* constraints requested by the task, not just the presence of individual constraints. The constraint characterizations described in Section 3.5 address this issue in a systematic manner.

Returning to Q1, we assess the impact of constraints on task scheduling delays. Our approach is to have the Treatment Specification in Figure 3.4 be "remove all constraints". Our evaluation metric is **normalized scheduling delay**, the ratio of the task scheduling delay in the baseline sub-workflow in which constraints are present to the task scheduling delay in the treatment sub-workflow in which all constraints are removed. Thus, a normalized scheduling delay of 1 means that there is no change from the baseline and hence constraints have no impact on task scheduling delays.

Figure 3.7 plots normalized scheduling delays for the three compute clusters. The horizontal axis is the raw trace file used as input for the experiment (see Figure 3.4). The vertical axis is the normalized scheduling delay, and there are separate bars for each task type. Observe that **the presence of constraints increases task scheduling delay by a factor of 2 to 6**. In absolute units, this often means tens of minutes of additional task wait time. The reason for this additional wait time is readily explained by examining the supply of machine resources that satisfy constraints and the task demand for these resources. For example, scheduling delays are smaller for tasks that request the first two *num_cpus* constraints (c5.[1-2].*) compared with tasks that request the *clock_speed* constraints (c7.*.*). This is because: (a) there are more machine resources that satisfy c5.[1-2].* than those that satisfy c7.*.* (see Figure 3.5); and (b) the task demand for c7.*.* is much greater than that for c5.[1-2].* (see Figure 3.6).

From the foregoing, we conclude that the presence of constraints dramatically increases task scheduling delays in the compute clusters we study. The degree of impact does, however, depend on the load on the compute cluster. Our analysis focuses on periods of heavy load since it is during these times that problems arise. During light loads, constraints may have little impact on task scheduling delays. However, our experience has been that if compute clusters are lightly loaded, then administrators remove machines to reduce costs. This results in much heavier loads on the remaining machines, and hence a much greater impact of constraints on task scheduling delays.

In this and the next section, we do not report results for resource utilization because our experiments do not reveal significant changes in resource utilization due to constraints. This is likely because only a small fraction of tasks are unable to schedule due to task placement constraints, and the impact of constraints on utilization is modest. However, the delays encountered by tasks with constraints can be quite large, and hence the impact of constraints on task average scheduling delays can be large.

3.4 Modeling Performance with Constraints

This section provides insight into how constraints impact task scheduling delays. Our approach is motivated by insights from queuing theory [107], in particular, that scheduling delays increase with resource utilization.

Our initial hypothesis is that task scheduling delays can be explained by

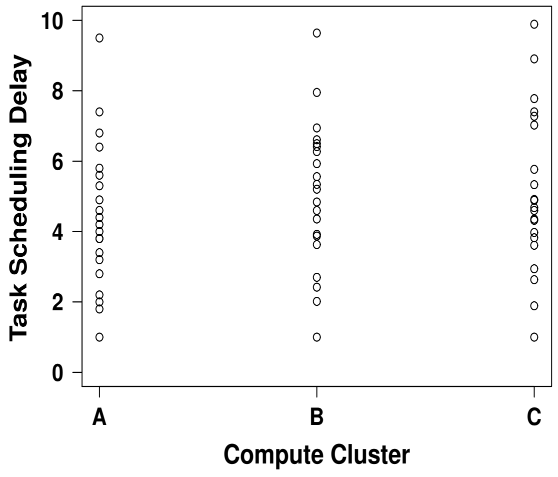


Figure 3.8. Scheduling delays for tasks with a single constraint. Each point is the average task scheduling delay for 10,000 type 1 tasks with one of the total 21 constraints we study.

average resource utilization without considering constraints. To test this hypothesis, we conduct studies in which we add to the raw trace 10,000 Type 1 tasks with small resource requirements. We restrict ourselves to TaskType 1 because of their importance. The added tasks have small resource requirements to ensure that they would schedule if there are no task placement constraints. Also, since the tasks have minimal resource requirements, they do not affect resource utilization; hence, for a single compute cluster, tasks with different constraints observe the same average resource utilization. We conduct a total of 315 studies, one study for each of the 15 raw trace files and each of the 21 constraints that we consider. Then, for each constraint c and each compute cluster, we average the delays observed in the five benchmarks of the compute cluster. From this, we calculate normalized task scheduling delays in the same way as is described in Section 3.3.

Figure 3.8 plots the results for each compute cluster. Recall, that the experiments are structured so that tasks scheduled on a compute cluster see the same average resource utilization regardless of the task constraint. Thus, if resource utilization without constraints is sufficient to explain task scheduling delays, then the points for a compute cluster should be grouped closely around a single value of task scheduling delay. However, this is not the case. Instead, we see a wide dispersion of points for each of the compute clusters. We conclude that resource utilization by itself cannot explain task scheduling delays if there are task placement constraints.

The foregoing motivates the need to extend the concept of resource utilization to include constraints. We use the term **effective utilization** to refer to the resource utilization seen by a task with consideration to its constraints. Modeling effective resource utilization allows us to answer the question **Q2: "Which constraints most impact task scheduling delays?"** And, answering this question is crucial to determine actions to take to reduce task scheduling delays by modifying applications codes and/or changing machine configurations.

We proceed based on insights obtained from Figure 3.1. Rather than computing effective utilization directly, we compute the ratio of effective utilization to average utilization. Our metric is the **Utilization Multiplier (UM)**. UM is computed for a particular constraint c and a resource r. In our analysis, r is in the resource set $\{\text{CPU, memory, disk}\}$. Computing separate values of UM for each r allows us to account for heterogeneous machine configurations.

We construct UM by considering both the task demand imposed by requests for constraint c and the supply of machine resources for which c is satisfied. We begin with the task demand. Let d_{cr} be the demand for resource r that is seen by tasks that request constraint c, and let d_r be the total demand for resource r across all tasks. We use the superscript D to indicate a demand metric. So, f_{cr}^D is the fraction of the demand for resource r due to tasks requesting constraint c:

$$f_{cr}^D = \frac{d_{cr}}{d_r}.$$

Next, we analyze the supply of machine resources. Let s_{cr} be the capacity of resource r on machines that satisfy constraint c. (Note that machine capacity is the "raw" supply without consideration for demands from other tasks.) Let s_r denote the total capacity of resource r on all machines. We use the superscript S to denote a supply metric. So, f_{cr}^S is the fraction of total resources r that satisfy constraint c:

$$f^S_{cr} = \frac{s_{cr}}{s_r}.$$

It is more convenient to consider the vector of resources rather than resources individually. Thus, we define for n resources,
$\mathbf{d}_c = (d_{c1}, \cdots, d_{cn})$, $\mathbf{s}_c = (s_{c1}, \cdots, s_{cn})$,
$\mathbf{d} = (d_1, \cdots, d_n)$, $\mathbf{s} = (s_1, \cdots, s_n)$,
$\mathbf{f}^D_c = (f^D_{c1}, \cdots, f^D_{cn})$ and $\mathbf{f}^S_c = (f^S_{c1}, \cdots, f^S_{cn})$.
Note that $\mathbf{f}^S_c, \mathbf{f}^D_c \leq \mathbf{1}$, where $\mathbf{1}$ is the unit vector with dimension n.

UM is the ratio of the fraction of the resource demand for a constraint to its supply. That is:

$$u_{cr} = \frac{f^D_{cr}}{f^S_{cr}}. \tag{3.1}$$

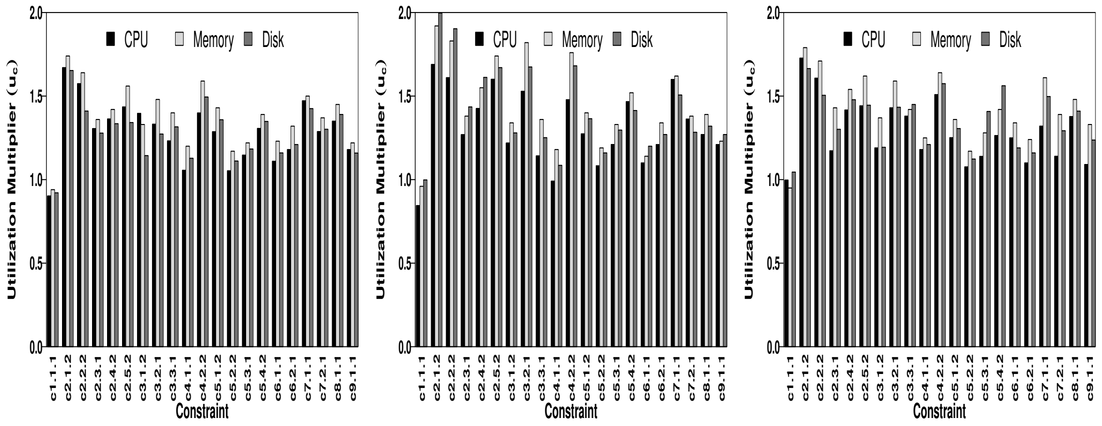
In vector notation, this is $\mathbf{u}_c = \frac{\mathbf{f}^D_c}{\mathbf{f}^S_c}$ where the division is done element by element. In what follows, all vector operations are done element by element.

$\mathbf{u}_c$ provides a number of insights. First, consider a constraint c that is requested by all tasks and is satisfied by all machines. Then, $\mathbf{f}^S_c = \mathbf{1} = \mathbf{f^D_c}$, and so $\mathbf{u}_c = \mathbf{1}$. That is, if UM is 1, then the constraint has no impact on the resource utilization seen by tasks with the constraint. In general, we expect that constraints limit the machines on which a task can run. This implies that $\mathbf{f}^S_c < \mathbf{1}$. There are some Google compute clusters in which most resource demands come from tasks that request constraints. For these compute clusters, it is likely that for one or more c $\mathbf{f}^D_c > \mathbf{f}^S_c$, and so $\mathbf{u}_c > \mathbf{1}$. That is, in this case, a task requesting c sees larger than the average resource utilization. On the other hand, in some Google compute clusters, tasks request a constraint c that will place the tasks on less desirable (e.g., older) machines thereby reducing effective resource utilization. Such a strategy works if $\mathbf{f}^S_c > \mathbf{f}^D_c$, such that $\mathbf{u}_c < \mathbf{1}$.

$\mathbf{u}_c$ can also be interpreted as an adjusted resource utilization, which motivates our phrase "effective resource utilization." Let ρ_c be the vector of resource utilization for machines that satisfy constraint c. That is, $\rho_c = \frac{\mathbf{d}_c}{\mathbf{s}_c}$. Let ρ be the vector resource utilization for all machines, and so $\rho = \frac{\mathbf{d}}{\mathbf{s}}$. Thus,

$$\mathbf{u}_c = \frac{\mathbf{f}_c^D}{\mathbf{f}_c^S} = \frac{\mathbf{d}_c}{\mathbf{d}}\frac{\mathbf{s}}{\mathbf{s}_c} = \frac{\mathbf{d}_c}{\mathbf{s}_c}\frac{\mathbf{s}}{\mathbf{d}} = \frac{\rho_c}{\rho}.$$

That is, the utilization of resource r seen by tasks that request constraint c is a factor of u_{cr} larger than the average utilization of resource r.



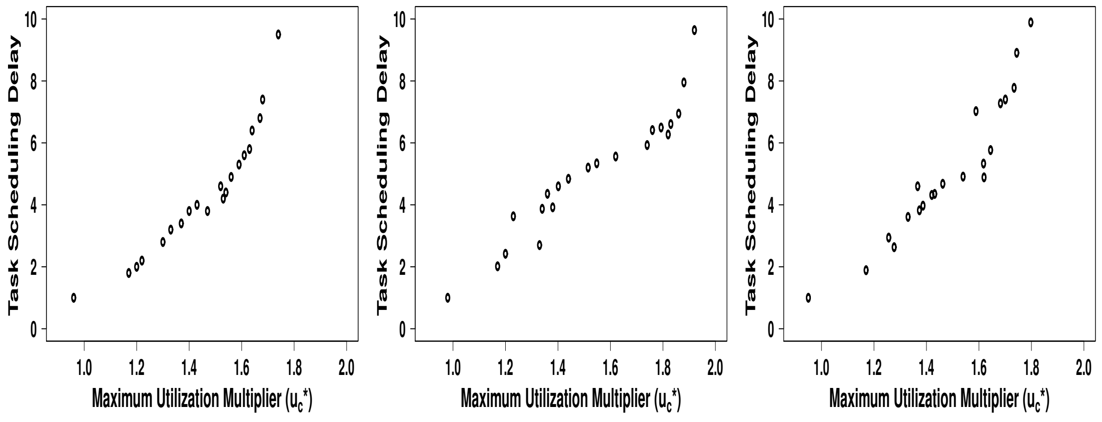
(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 3.9. Utilization Multiplier by resource for constraints.

Often, task scheduling delays are determined by the bottleneck resource rather than the entire resource vector. The bottleneck resource is the resource that has the largest utilization. Thus, we define the **maximum UM**, denoted by u_c^*, to be the scalar

$$u_c^* = max_r(u_{cr}). \tag{3.2}$$

Figure 3.9 shows $\mathbf{u}_c$ by CPU, memory, and disk for Type 1 tasks. Although the values of $\mathbf{u}_c$ vary from compute cluster to cluster, there is a general consistency in the relative magnitude of $\mathbf{u}_c$. For example, $\mathbf{u}_{c.1.1.1}$ is consistently smaller than the other constraints, and $\mathbf{u}_{c.2.1.2}$ is consistently one of the largest values. For the most part, $\mathbf{u}_c > \mathbf{1}$. The one exception is c1.1.1 where there are a few instances in which $\mathbf{u}_c \approx 1$. At a first glance, this is surprising since Figure 3.6 shows that a large fraction of tasks request c1.1.1. However, UM is the ratio of the demand for resources with c to the supply of these resources. From Figure 3.5, we know that there is a large supply of machine resources that satisfy c1.1.1. Hence, $\mathbf{u}_{c.1.1.1}$ is small.



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 3.10. Impact of maximum utilization multiplier on normalized task scheduling delay.

We return to the experiment described earlier in this section to see if UM provides a better explanation of task scheduling delays than average resource utilization. Figure 3.10 captures the relationship between u_c^* and normalized task scheduling delay. Note that for the most part, **UM provides a simple model of the performance impact of constraints in that task scheduling delays increase with maximum UM (u_c^*).** The relationship is linear for smaller values of u_c^*, but increases faster than linearly for larger u_c^*. Such curves are common in queuing analysis [107].

Figure 3.10 indicates that if we order constraints by UM, then we have also ordered constraints by their impact on task scheduling delays. From Figure 3.9, we see that in all three compute clusters, the constraint with the largest UM (and hence the largest impact on task scheduling delay) is c2.1.2, the first *num_cores* constraint. For compute cluster A, the constraints with the second and third largest impact on scheduling delays are *clock_speed* (c7.1.1) and *min_disks* (c4.2.2) constraints respectively. The situation is a bit different for compute cluster B. Here, the second and third largest scheduling delays are due to *max_disks* (c3.2.1) and *min_disks* (c4.2.2) constraints.

Much insight can be obtained from the simple model that task scheduling delays increase with UM. For example, consider the problem of resolving large scheduling delays for a task that has small resource requirements and many task placement constraints. We first compute the UM of the task's constraints in the compute cluster in which the task has long scheduling delays. We focus on the constraints with the largest u_c^*. It may be that some of these constraints can be eliminated from the task's request, especially if the constraints are actually preferences rather than requirements. However, if constraints with large u_c^* cannot be eliminated, an alternative is to find another compute cluster in which u_c^* is smaller. Such a cluster will either have less demand for machine resources that satisfy c or a larger supply of such resources.

3.5 Benchmarking with Constraints

This section investigates how to synthesize constraints to produce representative performance benchmarks. There are two parts to this: (1) characterizing task constraints and machine properties; and (2) incorporating synthetic constraints and properties into performance benchmarks.

3.5.1 Constraint Characterization

To characterize task placement constraints, we must address both tasks and machines. We use statistical clustering[1] to construct groups of tasks and machines, a commonly used approach in workload characterization [40]. A **task statistical cluster** is a group of tasks that are similar in terms of constraints that the tasks request. A **machine statistical cluster** is a group of machines that are similar in terms of constraints that the machines satisfy. Tasks and machines belong to exactly one statistical cluster.

Task and machine statistical clusters have a common structure, and so the following definitions apply to both. To this end, we use the term *entity* to refer to both task and machine. A statistical cluster has two metrics. The first metric is the scalar **cluster occurrence fraction**. This is the fraction of all entities (of

[1]Unfortunately, the statistics and cloud computing communities both use the term cluster but with very different semantics. We distinguish between these semantics by using the phrases "statistical cluster" and "compute cluster".

the same type in the same compute cluster) that are members of the statistical cluster. The second metric is the **constraint frequency vector**. This vector has one element for each constraint c. The vector element for c is the scalar **constraint occurrence fraction**, which is the fraction of entities in the statistical cluster that "have" the constraint c. For tasks, "have" means that the task requests the constraint, and for machines "have" means that the machine satisfies the constraint.

We use the term **machine constraint characterization** to refer to the set of machine statistical clusters. The **task constraint characterization** is defined analogously. A **constraint characterization** is the combination of the machine constraint characterization and the task constraint characterization.

We construct machine statistical clusters by using a binary feature vector. For each machine, the i-th element of its feature vector is 1 if the machine satisfies the i-th constraint; otherwise, the element is 0. The elements of the feature vector are indexed by descending value of the maximum UM of the constraint. That is, $u^*_{c_i} \geq u^*_{c_{i+1}}$. Initially, we used k-means [108] algorithm to construct machine statistical clusters. However, this proved to be unnecessarily complex for our data. Rather, it turns out that a simple approach based on sorting works extremely well. Our approach starts by sorting the machine feature vectors lexicographically. We form machine statistical clusters by placing together machines that are adjacent in the sorted sequence as long as no element in their constraint frequency vector differs by more than 0.05. The choice of 0.05 is empirical; it is based on the trade-off between the goals of having few statistical clusters and having statistical clusters with homogeneous machine properties. The result is four machine statistical clusters for each of the three compute clusters that we study.

For task statistical clusters, we also use a binary feature vector. Here, element i is 1 if the task requests constraint c_i; otherwise, the vector value is 0. Task statistical clusters are constructed using k-means on the task feature vector. K-means is used to construct task statistical clusters because many values in the task constraint frequency vectors are close to 0.5 and this precludes using simple clustering algorithms as we did with constructing machine statistical clusters.

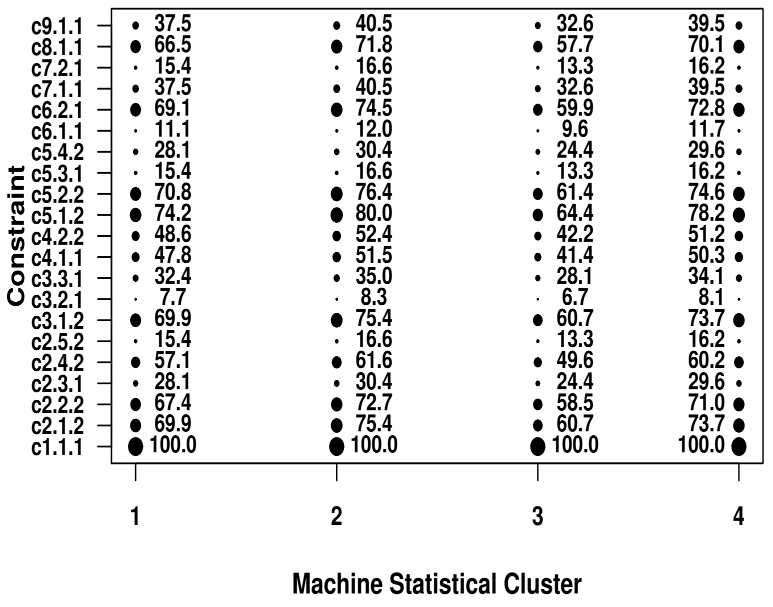


Figure 3.11. Machine statistical clusters for compute cluster A.

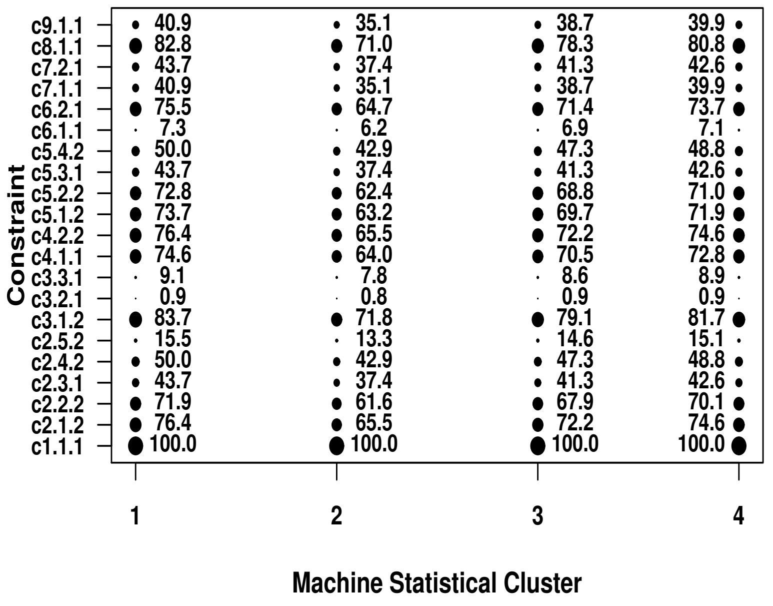


Figure 3.12. Machine statistical clusters for compute cluster B.

Figure 3.11 and Figure 3.12 display the constraint frequency vector for machine statistical clusters A and B, respectively. The results for compute cluster C are similar to those for A and B. More complete data can be found in [109].

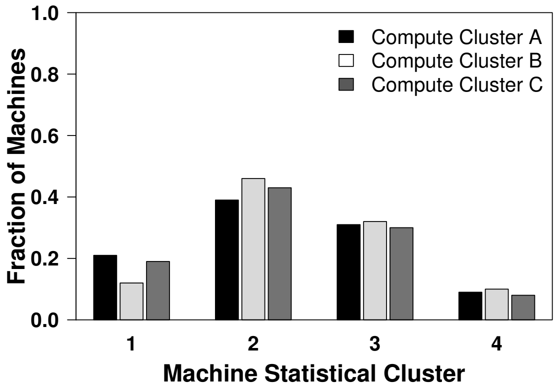


Figure 3.13. Occurrence fraction for machine statistical clusters.

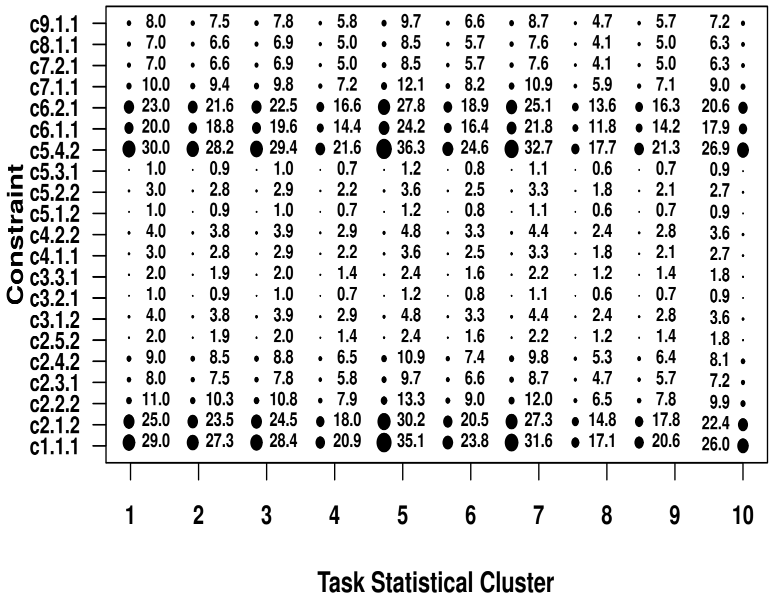


Figure 3.14. Task statistical clusters for compute cluster A.

The horizontal axis is the machine statistical cluster, and the vertical axis is the constraint. Each point is sized in proportion to the fraction of machines in the statistical cluster that satisfy the constraint (with the actual value marked alongside). For example, in Figure 3.11, about 74.2% of the machines that belong to statistical cluster 1 satisfy constraint c5.1.2. We see that the machine statistical clusters are fairly consistent between the compute clusters. Indeed, even though there are thousands of machines in each compute cluster, the variation in machine properties can largely be explained by only four statistical clusters.

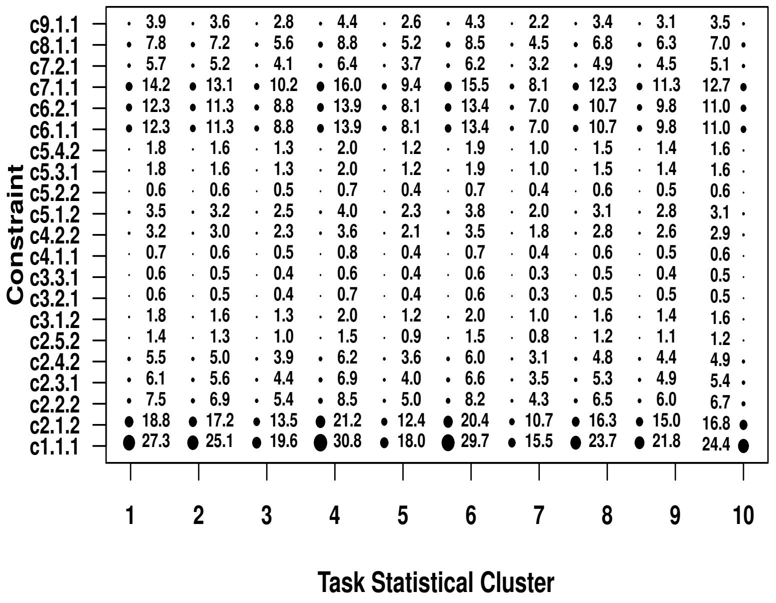
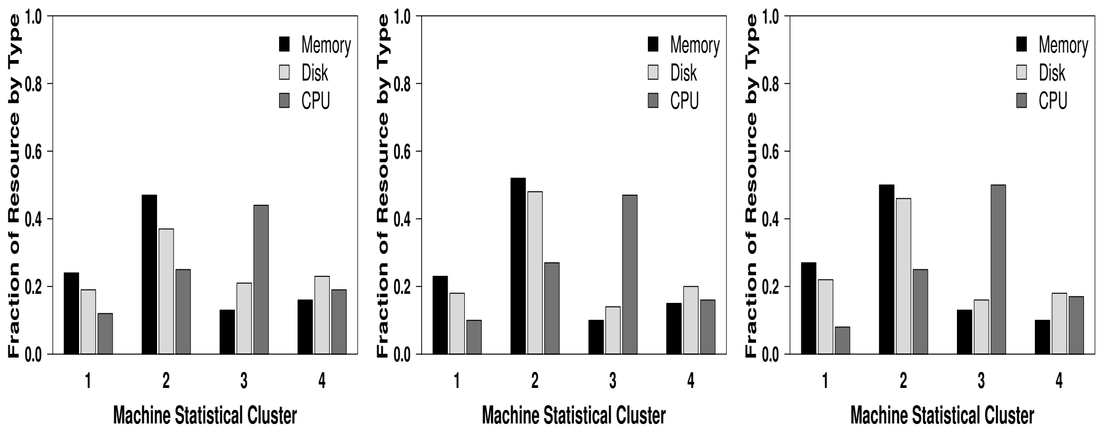


Figure 3.15. Task statistical clusters for compute cluster B.

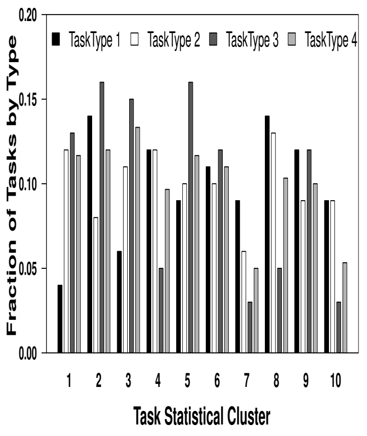


(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

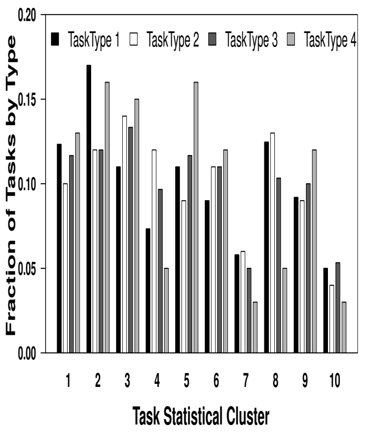
Figure 3.16. Resource distribution for machine statistical clusters.

Figure 3.13 shows the occurrence fractions of the machine statistical clusters in the three Google compute clusters. The horizontal axis is the machine statistical cluster and the vertical axis is the fraction of machines that belong to the statistical cluster. For example, in Figure 3.13, approximately 20% of the machines in compute cluster A belong to machine statistical cluster 1.

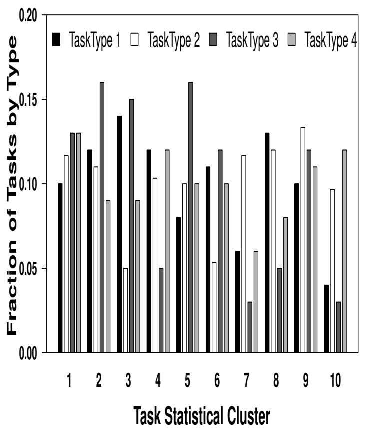
Figure 3.14 and Figure 3.15 show the constraint frequency vectors of the task



(a) Compute Cluster A

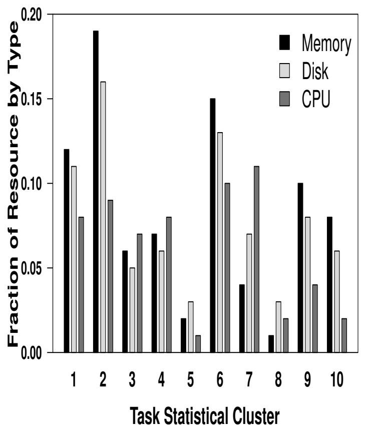


(b) Compute Cluster B

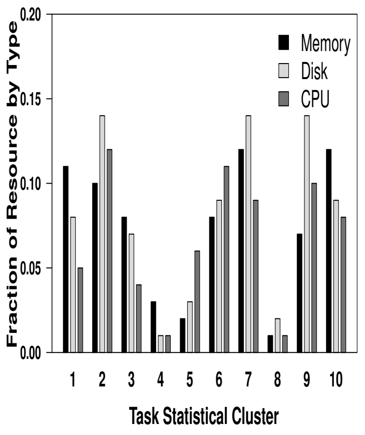


(c) Compute Cluster C

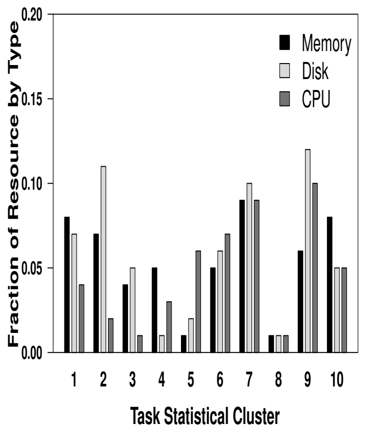
Figure 3.17. Occurrence fraction for task statistical clusters.



(a) Compute Cluster A



(b) Compute Cluster B



(c) Compute Cluster C

Figure 3.18. Resource distribution for task statistical clusters.

statistical clusters for compute clusters A and B, respectively. The figure is structured in the same way as for the machine statistical clusters in Figure 3.11 and Figure 3.12. The horizontal axis is the task statistical cluster; the vertical axis is the constraint; each point is sized in proportion to the fraction of tasks in the statistical cluster that request the constraint (with the actual value marked alongside). The task statistical clusters are more difficult to interpret than the machine statistical clusters because: (a) many constraints have values closer to 0.5; (b) there is more similarity between the statistical clusters; and (c) it is difficult to relate task clusters in one compute cluster to task clusters in another compute cluster. Figure 3.16 displays the fraction of resources supplied by the machine statistical clusters. The horizontal axis is the machine statistical cluster, and the vertical axis is the fraction of resources by resource type. For instance, in Figure 3.16 (a), approximately 25% of the memory capacity in compute cluster A is on machines belonging to statistical cluster 1.

Figure 3.17 displays the occurrence fractions of the task statistical clusters by task type. The horizontal axis is the task statistical cluster, and the vertical axis is the fraction of tasks (by type) that belong to the statistical cluster. For instance, in Figure 3.17 (a), approximately 4% of Type 1 tasks belong to task statistical cluster 1. To provide further insight into the nature of tasks in clusters, Figure 3.18 shows the distribution of resources demanded (by task type) for the 10 task statistical clusters. These data can be used in performance benchmarks to specify resource requirements that are representative of Google compute clusters.

3.5.2 Extending Performance Benchmarks

Next, we show how to extend existing performance benchmarks to incorporate task constraints and machine properties. First, we detail how to characterize and synthesize representative task constraints and machine properties. Then, we show that how to incorporate this synthesis into existing performance benchmarks such as those described in [110, 56].

Algorithm 1 Extend an existing benchmark to include task placement constraints.

```
Require: Existing benchmark that generates machine resource capacities and task
      resource requirements.
1: if event = benchmark initialization then
2:    Assign properties to machines  using Algorithm 2.
3: end if
4: if event = task arrival then
5:    Assign constraints to the task  using Algorithm 3.
6: end if
7: if event = task is a candidate to run on a machine then
8:    Determine if the machine satisfies the constraints required by the task using
      Algorithm 4.
9: end if
```

Algorithm 1 describes the logic added to such an existing benchmark in order to

Algorithm 2 : Assign properties to a machine.

```
Require: MachineConstraintFrequencyVector (Figure 3.11 and Figure 3.12),
    MachineOccurrenceFraction (Figure 3.13), Machine
1: cluster = randomly choose cluster weighted by MachineOccurrenceFraction
2: mcfv = MachineConstraintFrequencyVector for cluster
3: for constraint in mcfv do
4:    if random(0,1) $\leq$ mcfv[constraint].ConstraintOccurrenceFraction then
5:       Machine.add(property that satisfies the constraint)
6:    end if
7: end for
```

Algorithm 3 : Assign task placement constraints to a task.

```
Require: TaskOccurrenceFraction(Figure3.17), TaskConstraintFrequencyVector
    (Figure 3.14 and Figure 3.15), Task
1: cluster = randomly choose cluster weighted by TaskOccurrenceFraction
2: tcfv = TaskConstraintFrequencyVector for cluster
3: for constraint in tcfv do
4:    if random(0,1) $\leq$ tcfv[constraint].ConstraintOccurrenceFraction then
5:       Task.add(constraint)
6:    end if
7: end for
```

Algorithm 4 : Determine if a task can be run on a machine. Returns TRUE if Task can run on Machine.

```
Require: Task, Machine
1: for constraint in Task.constraints() do
2:    if constraint is not in Machine.constraints() then
3:       return FALSE
4:    end if
5: end for
6: return TRUE
```

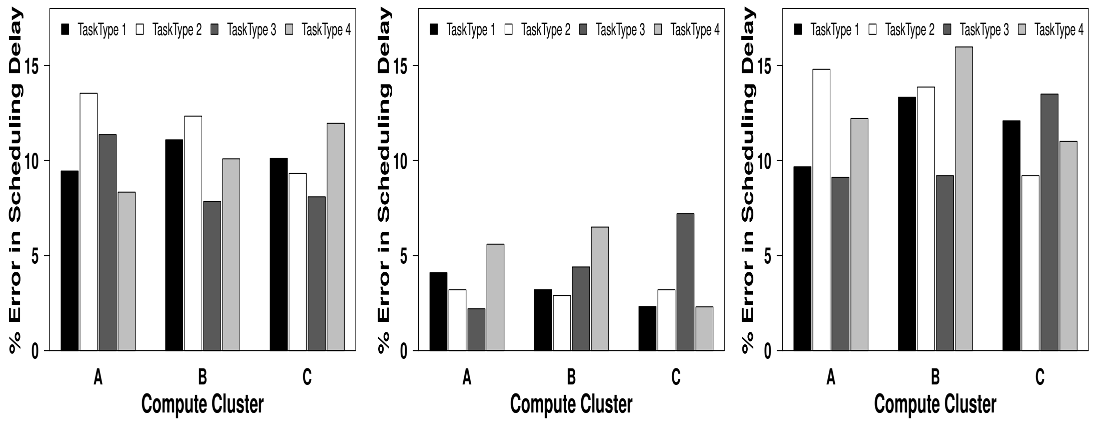
incorporate representative constraints. This logic is event-based, with processing done when the following events occur: (a) the benchmark is initialized; (b) a task arrives; and (c) a task is a candidate to be placed on a machine.

Algorithm 2 details how to assign properties to a machine. In step 1, a machine statistical cluster is chosen randomly based on the occurrence fractions of machine statistical clusters (e.g., Figure 3.13). The algorithm uses the constraint frequency vector of the chosen machine statistical cluster (e.g., Figure 3.11 and Figure 3.12) and a random number generator to select constraints that the machine must satisfy. The structure of constraints in Figure 3.2 makes it trivial to specify a property that satisfies a constraint. For example, if the constraint is "*num_cores* ≥ 2", then the machine is assigned the value of 2 for its num_cores attribute. In general, there can be logical inconsistencies in the properties inferred in this manner. However, this problem does not arise for the set of constraints of the four machine statistical clusters for the compute clusters that we study.

Algorithm 3 assigns constraints to tasks in a manner similar to what Algorithm 2 does for machines. A task statistical cluster is chosen randomly based on the occurrence fractions of task statistical clusters (e.g., Figure 3.17). Then, the constraint frequency vector of the chosen task statistical cluster (e.g., Figure 3.14 and Figure 3.15) is used to assign constraints. Note that the logic assumes that the task type is known. If this is not the case, a task type can be chosen randomly using the distributions in Figure 3.3.

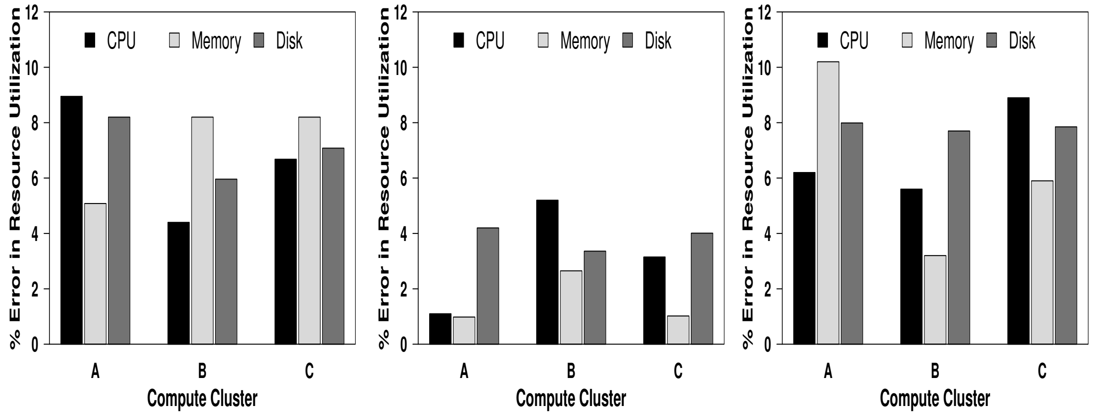
Algorithm 4 describes the logic added to an existing compute cluster scheduler to take into account task constraints. Before a task is assigned to a machine, the cluster scheduler calls Algorithm 4. The algorithm returns true only if the machine satisfies all of the constraints required by the task.

How accurately do the foregoing algorithms reproduce the performance characteristics of Google compute clusters? To answer this question, we construct experiments by changing the Treatment Specification block in Figure 3.4 to synthesize constraints for machines and tasks using Algorithm 1. Specifically, for all machines in the trace, we remove the machine's properties and then apply Algorithm 2 to generate synthetic constraints that in turn determine the properties that are assigned to the machines. Similarly, for all tasks in the trace, we remove its constraints and then apply Algorithm 3 to generate synthetic constraints that replace the task's constraints in the raw trace. Each study produces two metrics: average error in task scheduling delay and average error in machine resource utilization. The error in task scheduling delays is computed as the percent deviation of the average task scheduling delay in the treatment (synthesized constraints) from the average scheduling delay in the baseline (constraints in the raw trace). The error in compute cluster resource utilization is



(a) Task Constraint (b) Machine Property (c) Constraint & Property

Figure 3.19. Percent error in task scheduling delay resulting from using synthetic task constraints and/or machine properties.



(a) Task Constraint (b) Machine Property (c) Constraint & Property

Figure 3.20. Percent error in compute cluster resource utilization resulting from using synthetic task constraints and/or machine properties.

computed similarly.

Figure 3.19 shows the results of our evaluation of the workload characterizations for task scheduling delays. Figure 3.19(a) displays the errors introduced if we use synthetic constraints for tasks and the actual machine properties. We see that synthesizing task constraints introduces an error of approximately 8%. Figure 3.19(b) displays the errors in task scheduling delays if only machine constraints are synthesized. Here, we see an average error of around 5%. Figure 3.19(c) displays the errors if both task and machine constraints are synthesized. The average error is approximately 13%.

Figure 3.20 analyzes the errors in compute cluster resource (CPU, memory and disk) utilization introduced by using synthetically generated constraints. Figure 3.20(a) displays the results when we synthesize task constraints and use the actual machine properties. We see that this introduces an error of around 6%. Figure 3.20(b) shows the errors resulting from synthesizing machine constraints and using the actual task constraints. Here, we see an average error of around 3%. In Figure 3.20(c), we see that the average error is approximately 5% if both task and machine constraints are synthesized.

From these results, we conclude that **our constraint synthesis produces representative workloads for Google compute clusters**. Specifically, synthetic workloads generated using our constraint characterizations result in task scheduling delays that differ by an average of 13% from what is produced by using the production machines/tasks. And, our approach produces machine resource utilization that differ by an average of 5% from the production machines/tasks.

Although our methodology for building performance benchmarks that incorporate constraints is illustrated using data from Google compute clusters, the methodology is not specific to Google. Our characterization of constraints for machines and tasks is done in a general way using clustering, cluster occurrence fractions, and cluster constraint frequency vectors. Our approach towards incorporating these characterizations into performance benchmarks is also quite general, as described in Algorithm 1 - Algorithm 4.

3.6 Chapter Summary

There has been much prior work on task scheduling that considers resource requirements that address *how much* resource tasks consume. This chapter addresses the performance impact of task placement constraints. Task placement constraints impact *which* resources tasks consume. Task placement constraints, such as characteristics specified by the Condor ClassAds mechanism, provide a way to deal with machine heterogeneity and diverse software requirements in compute clusters. Our experience at Google suggests that task placement constraints can have a large impact on task scheduling delays.

The work presented in this chapter develops a methodology that addresses the performance impact of task placement constraints in representative cloud clusters. We show that in Google compute clusters, constraints can increase average task scheduling delays by a factor of 2 to 6, which often means tens of minutes of additional task wait time. To understand why, we introduce a new metric, the Utilization Multiplier (UM), that extends the concept of resource utilization to include constraints. We show that task scheduling delays increase with UM for the tasks that we study. We also show how to characterize and generate representative task constraints and machine properties, and how to incorporate synthetic constraints and properties into existing performance benchmarks. Applying our approach to Google compute clusters, we find that our constraint characterizations accurately reproduce production performance characteristics. Specifically, our constraint synthesis produces benchmark results that differ from production workloads by an average of 13% in task scheduling delays and by an average of 5% in machine resource utilizations.

Although our data is obtained from Google compute clusters, the methodology that we develop is general. In particular, the UM metric applies to any compute cluster that employs a ClassAds style of constraint mechanism. We look forward to seeing data that provides insights into the performance impact of task placement constraints in other compute clusters.

Chapter 4

Resource Management in Big Data MapReduce Cloud Clusters

4.1 Introduction

Google MapReduce [14] has emerged as an important cloud activity for massive distributed data processing. Several academic and commercial organizations [111] like Facebook, Microsoft and Yahoo! use an open source implementation called Hadoop MapReduce [15] to effectively process their huge set of data. In public utility clouds like Amazon EC2, MapReduce is gaining prominence with customized services such as Elastic MapReduce [29] for providing the desired backbone for efficient Internet-scale data analytics.

Currently, resource allocation in Hadoop MapReduce is done at the granularity of fixed-size resource splits of the nodes, called *slots*. A slot is a basic unit of resource allocation, representing a fixed proportion of multiple shared resources (like CPU, memory and disk) on a machine. Only one map/reduce task can run per slot at a time. The primary advantage of a slot is its simplicity and ease of implementation of the MapReduce paradigm.

The slot-based resource allocation in Hadoop has three main disadvantages. The first downside is related to the fixed-size and static definition of a slot.

Currently, in Hadoop framework, a node is configured with a fixed number of slots, which are statically [1] estimated in an ad-hoc manner irrespective of the machine's dynamically varying resource capacities. A slot is too coarse an allocation unit to represent the actual resource demand of a task, leading to wastage of individual resources when multiple of those are paired together in a slot. We observed in our experiments that slot-level allocations can lead to scenarios, where some of the resources are under-utilized, while others become bottlenecks. Likewise, analysis on a 2000-node Hadoop cluster at Facebook [17] has shown both the under and over cluster utilization due to significant disparity between tasks resource demands and slot resources.

The second and third problems are attributed to the lack of isolation and uncoordinated allocation of resources across the nodes of a Hadoop cluster, respectively. Although there is a form of 'slot-level' sharing of resources across jobs that is enforced by the Hadoop fair scheduler [69], there is no implicit partitioning of resources across jobs. This can lead to negative interference of resources among jobs. For example, a recent study on a Microsoft production MapReduce cluster indicates high contention for dynamic resources like CPU and memory [18]. Further, multiple nodes running different jobs are agnostic of their resource demands and contentions. Lack of global coordination in the management of multiple resources across the nodes can lead to situations, where local resource management decisions contradict with each other. Thus, when there are multiple concurrently executing MapReduce jobs, lack of isolation and uncoordinated sharing of resources can lead to poor performance.

Towards this end, this chapter makes the following contributions:

- This chapter presents the design and implementation of a novel resource management framework, *MROrchestrator*, that provides fine-grained, dynamic and coordinated allocation of resources to MapReduce jobs. Based on the run-time resource profiles of tasks, *MROrchestrator* builds on-line resource estimation models for on-demand allocations. *MROrchestrator* is a software

[1]Based on the Hadoop code base [112], the number of slots per node is implemented as the minimum amount of the resource in the tuple {C-1, (M-2)/2, D/50}; where C = number of cores, M = memory in GB, D = disk space in GB, per node.

layer that assumes or requires no changes to the Hadoop framework, making it a simple, flexible, portable and platform generic design architecture.

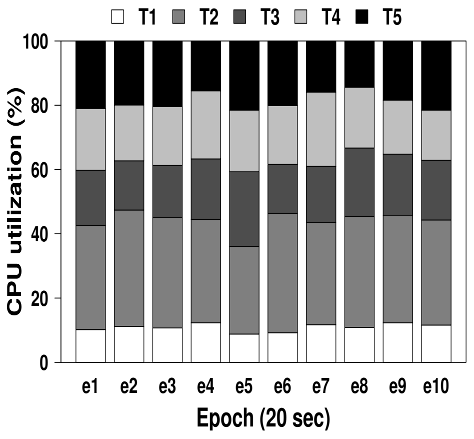
- Detailed experimental analysis on two 24-node native and virtualized Hadoop clusters demonstrate the benefits of *MROrchestrator* in terms of reducing job finish times and boosting resource utilization. Specifically, *MROrchestrator* achieves up to 38% reduction in job completion times and up to 25% increase in resource utilization, compared to the slot-based resource allocation schemes.
- *MROrchestrator* is complementary to the contemporary resource scheduling managers like Mesos [72] and Next Generation MapReduce (NGM) [113]. It can be augmented with these frameworks to boost system performance. Results from the integration of *MROrchestrator* with Mesos and NGM demonstrate up to 17% and 23% reduction in the job completion times, respectively. In terms of resource utilization, there is a corresponding increase of 12% (CPU), 8.5% (memory); 19% (CPU), 13% (memory), respectively.

The rest of this chapter is organized as follows: Section 4.2 outlines motivation for the problem addressed. The design and implementation details of *MROrchestrator* is described in Section 4.3. Section 4.4 presents the experimental platform and evaluation results. The chapter summary is outlined in Section 4.5.

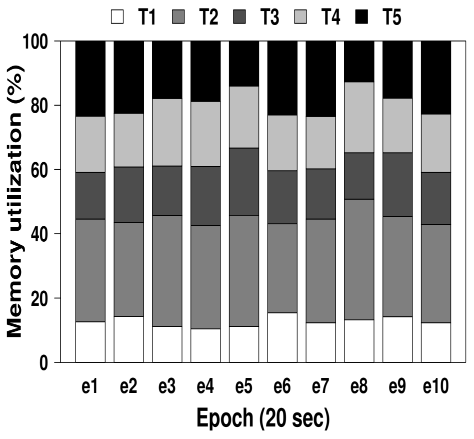
4.2 Motivation

4.2.1 Need for Dynamic Resource Allocation and Isolation

As stated above, slot-level resource allocation, which does not provide any implicit resource partitioning and isolation, leads to high resource contentions. Furthermore, Hadoop framework is agnostic to the dynamic variation in the run-time resource profiles of tasks across map and reduce phases, and statically allocates resources in a coarse grained slot unit. This leads to wastage of resources since not all contained in a slot are proportionally utilized.

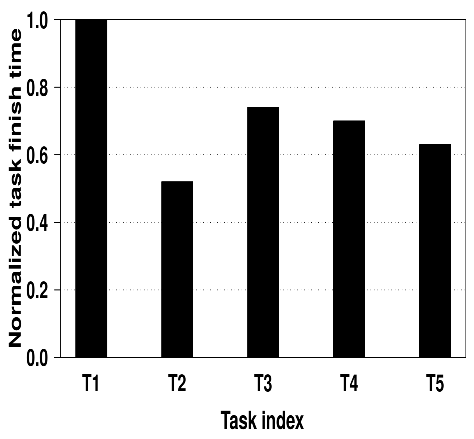


(a) CPU utilization

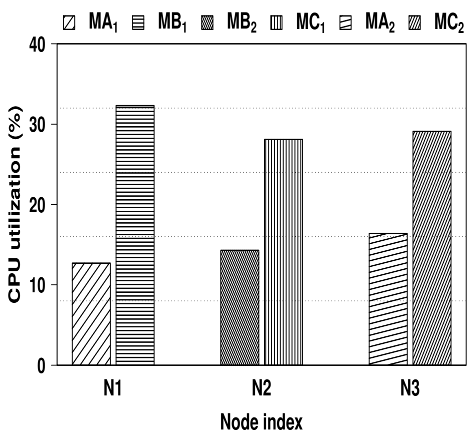


(b) Memory utilization

Figure 4.1. Illustration of the variation in the resource usage of the constituent tasks of a `Sort` MapReduce job.



(a) Finish time of tasks



(b) Lack of global coordination

Figure 4.2. Illustration of the variation in the finish times of the constituent tasks of a `Sort` MapReduce job (a), and global coordination problem [(b)]. Y-axis in plot (a) is normalized with respect to the maximum value.

We provide some empirical evidences to the aforesaid problems related to the slot-based resource allocations. We run a `Sort` MapReduce job with 20 GB text input on a 24-node Hadoop cluster (experimental details are described in Section 4.3.3.1). We observe variation in resource utilization and finish times across the constituent map/reduce tasks of this job. Figures 4.1(a) and 4.1(b) show the CPU and memory utilization variation for five concurrently running reduce tasks of `Sort` job on a node across 10 randomly selected epochs of their total run-time. From these figures, we can observe that multiple tasks have different resource utilization across these epochs. Some task (*t2*) gets more CPU and memory entitlements and consequently finish faster (see Figure 4.2(a)) when compared with the other concurrently executing tasks. This observation can stem from a variety of reasons – node heterogeneity, data skewness and network traffic, which are prevalent in a MapReduce environment [18]. Here, the variation in disk utilization of these tasks is low since most of the data resides in the memory of each node (20 GB data split across 24 nodes). Further, due to an inherent barrier between the map and reduce phase [14], such variation in resource usage and finish times is disadvantageous since the completion time of a MapReduce job is constrained by the slowest task.

4.2.2 Need for Global Resource Coordination

Cluster nodes hosting MapReduce jobs when unaware of the run-time resource profiles of constituent tasks can lead to poor system performance. We illustrate this aspect through an example. Consider 3 nodes N_1, N_2, N_3, executing 3 jobs A, B and C, with 2 map tasks each. That is, N_1 is shared by a map task of A and B, denoted as MA_1 and MB_1, N_2 is shared by a map task of B and C, denoted as MB_2 and MC_1, and N_3 is shared by a map task of A and C, denoted as MA_2 and MC_2. In this scenario, if we detect that MB_1 is hogging CPU allocation of MA_1, and change the CPU allocations between MB_1 and MA_1, we may not be able to improve job A's performance, because MC_2 may be contending with MA_2 for CPU. Also, MC_1 may be contending with MB_2 for CPU. Therefore, reducing MB_1's CPU allocation at N_1 (based on the local information) will only hurt B's performance without improving A's performance. The above scenario was observed in a simple experiment, where we ran `Sort` (A), `Wcount` (B) and `DistGrep` (C) jobs on a 24-node Hadoop cluster (details in Section 4.3.3.1). Figure 4.2(b) depicts the corresponding results. Such inefficiencies can be avoided with proper global coordination among all the cluster nodes.

4.3 Design and Implementation of *MROrchestrator*

Figure 4.3 shows the architecture of *MROrchestrator*. Its main components include a *Global Resource Manager (GRM)*, running on the JobTracker, and a *Local Resource Manager (LRM)*, running on each TaskTracker. The GRM consists of two sub-components: (i) a *Contention Detector* that dissects the cause of resource contention and identifies both resource-deficit and resource-hogging tasks; and (ii) a *Performance Balancer* that leverages the run-time resource estimations from each LRM to suggest the resource adjustments to each, based on the global coordinated view of all tasks running on TaskTrackers. The LRM also consists of the following two sub-components: (i) a *Resource Profiler* that collects and profiles the run-time resource usage/allocation of tasks at each *TaskTracker*; and (ii) an *Estimator* that constructs statistical estimation models of a task's run-time performance as a function of its resource allocations. The Estimator can have different performance estimation models. As with other resource managers like Mesos, we currently focus on CPU and memory as the two major resources [18] for dynamic allocation and isolation. We plan to address disk and network resources in the near future.

MROrchestrator performs the following two main functions: (i) Detection of the resource bottleneck; and (ii) Perform dynamic resolution of resource contention across tasks and nodes. We describe each of these functions in detail below:

Resource Bottleneck Detection: The functionalities of this phase can be summarized in two parts (denoted as steps **1** and **2** in Figure 4.3): **(1)** At regular intervals, the Resource Profiler in each LRM monitors and collects the run-time resource utilization and allocation information of each task using the *Hadoop profiler* [114]. This data is sent back to the Contention Detector module of GRM at JobTracker, piggy-backed with the heartbeat messages [15]. **(2)** On receiving these heartbeat messages from all the LRMs, the GRM can identify *which task* is experiencing bottleneck for *which resource*, on *which TaskTracker*, based on the following rationale. When different jobs, each with multiple map/reduce tasks,

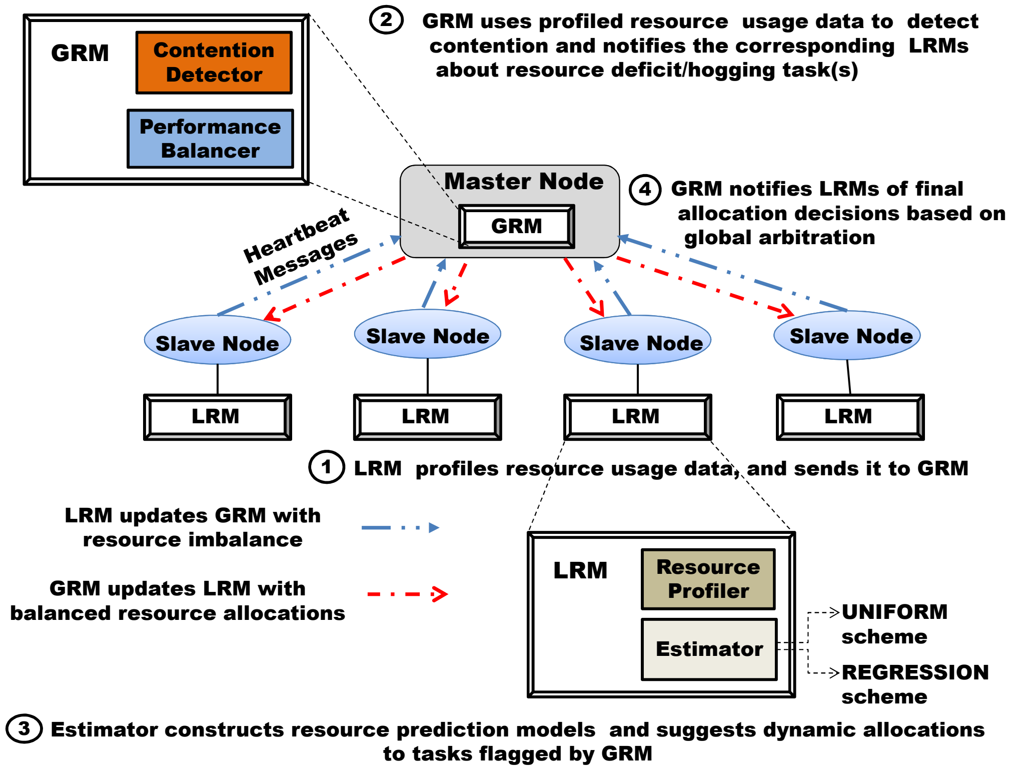

GRM
Contention Detector
Performance Balancer
2 GRM uses profiled resource usage data to detect contention and notifies the corresponding LRMs about resource deficit/hogging task(s)
Master Node
GRM
4 GRM notifies LRMs of final allocation decisions based on global arbitration
Heartbeat Messages
Slave Node
Slave Node
Slave Node
Slave Node
LRM
LRM
LRM
LRM
1 LRM profiles resource usage data, and sends it to GRM
LRM updates GRM with resource imbalance
GRM updates LRM with balanced resource allocations
LRM
Resource Profiler
Estimator
UNIFORM scheme
REGRESSION scheme
3 Estimator constructs resource prediction models and suggests dynamic allocations to tasks flagged by GRM

Figure 4.3. Architecture of *MROrchestrator*.

concurrently execute on a shared cluster, we expect similar resource usage profiles across tasks within each job. This assumption stems from the fact that tasks operating in either the map or reduce phase typically perform the same operations (map or reduce function) on similar input size, thus requiring identical amount of shared resources (CPU, memory, disk and network bandwidth). Due to practical factors like node heterogeneity, data skewness and cross-rack traffic, there is wide variation in the run-time resource profiles of tasks due to slot-based resource allocations in Hadoop. We exploit these characteristics here to identify potential resource bottlenecks and the affected tasks. For example, if a job is executing 6 map tasks across 6 nodes, and we see that the memory utilization of 5 of the 6 tasks on nodes 1–5 is close to 60%, but the memory utilization of only one of the tasks on node 6 is less than 25%, the GRM, based on its global view of all the nodes, should be able to deduce that the particular task is potentially memory deficit.

This approach might not work properly in some cases. For example, some outlier tasks may have very different resource demands and usage behavior, or because of workload imbalance, some tasks may get more share of input than the others. Thus, the resource profiles of such tasks may be quite deviant (although normal), and could be misinterpreted in this approach. In such scenarios, we adopt an alternative approach– we leverage the functionality of the JobTracker that can identify the straggler tasks [14] based on their progress score [74]. Since, resource contention is a leading cause for stragglers [18, 17], the GRM based on these two features can explicitly identify the potential resource contention induced straggling tasks from others.

Resource Bottleneck Mitigation: The functionalities of this phase can be explained using the remaining steps **3** and **4**, as shown in Figure 4.3. **(3)** After getting the information about both the resource deficit and resource hogging tasks from the GRM, the LRM at each TaskTracker invokes its Estimator module to estimate the task completion times (see Section 4.3.1 and Section 4.3.2). The Estimator also maintains mappings between task execution time and run-time resource allocations. The Estimator builds predictive models for the finish time of a task as a function of its resource usage/allocation. The difference between the estimated and the current resource allocation is the resource imbalance that has to be dynamically allocated to the resource-deficit task. This information is piggy-backed in the heartbeat messages from the respective LRM to the GRM. After receiving the resource imbalances (if any) from every LRM, the Performance Balancer module in GRM executes the following simple heuristic: the GRM at JobTracker uses the global view of all running tasks on TaskTrackers to determine if the requested adjustment in resource allocation is in *sync* (see Section 4.2.2) with other concurrent tasks (of the same phase and job) on other TaskTrackers. **(4)** Based on this global coordinated view, the GRM notifies each LRM of its approval of suggested resource adjustment, following which each LRM triggers the dynamic resource allocation.

4.3.1 Resource Allocation and Progress of Tasks

In Hadoop MapReduce, a metric called *progress score* is used to monitor the run-time progress of each task. For the map phase, it represents the fraction of the input data read by a map task. For the reduce phase, it denotes the fraction of the intermediate data processed by a reduce task. Since, all the tasks in a map or reduce phase perform similar operations, the amount of resources consumed by a task is assumed to be proportional to the input data. In turn, the amount of consumed resources will be proportional to the progress score of each task. Therefore, as proposed in [74], in the absence of any change in the allocated resource for a task, the predicted task finish time, $\hat{T}$ can be expressed as

$$\hat{T} = \frac{1 - ProgressScore}{ProgressScore} T \,, \tag{4.1}$$

where T represents the elapsed time of a task. However, by varying the resource allocated to a task, the remaining finish time, $\hat{T}$ of a task may change. $\hat{T}$ can thus be represented as

$$\hat{T} = \alpha \frac{1 - ProgressScore}{ProgressScore} T \,. \tag{4.2}$$

Depending on α (which represents the resource scaling factor), the estimated finish time can increase ($\alpha > 1$, indicating resource deficiency) or decrease ($\alpha < 1$, indicating excess resource). $\alpha = 1$ signifies the balanced task resource allocation. In order to control α, we propose two choices for the Estimator. These schemes form part of the Estimator module in LRM and are described below.

4.3.2 Estimator Design

The Estimator module in LRM is responsible for building the predictive models for tasks run-time resource usage/allocations. It can plug-in a variety of resource allocation schemes. We demonstrate two such schemes for *MROrchestrator*. The first scheme is called *REGRESSION*, which uses statistical regression models to obtain the estimated resource allocation for the next run epoch. The second scheme is called *UNIFORM*. It collects the past resource usage of all tasks in a phase, computes the *mean* usage across all these tasks and uses this value as the predicted resource allocation for the next epoch. Details of each scheme are described below.

4.3.2.1 REGRESSION scheme

This scheme determines the amount of resources (CPU and memory) to be allocated to each task at run-time, while considering the various practical scenarios that may arise in typical MapReduce clusters like node heterogeneity, workload imbalance, network congestion and faulty nodes [18]. It consists of two stages. The first stage inputs a time-series of progress scores of a given task and outputs a time-series of estimated finish times corresponding to multiple epochs in its life time. The finish time is estimated using Equation 4.1. The second stage takes as input a time-series of past resource usage/allocations across multiple epochs of a task's run-time (from its start time till the point of estimation). It outputs a statistical regression model that yields the estimated resource allocation for the next epoch as a function of its cumulative run-time. Thus, at any epoch in the life-time of a task, its predicted finish time is computed from the first model, and then the second model estimates the resource allocation required to meet the target finish time.

Separate estimation models are built for the CPU and memory profiles of a task. For the CPU profile, based on our experiments, we observed that the choice of a linear model achieves a good fit. For the memory profile, however, a linear model based on the training data from entire past resource usage history does not fit well, possibly due to the high dynamism in the memory usage of tasks. We use the following variant to better capture the memory footprints of the task – instead of using the entire history of memory usage, we only select training data over some recent time windows. The intuition is that a task has different memory usage across its map and reduce phases, and also within a phase. Thus, training data corresponding to recent time history is more representative of a task's memory profile. The memory usage is then captured by a linear regression model over the recent past time windows.

The number of recent time windows across which the training data is collected

Algorithm 5 Computation of the estimated resource (at LRM).

```
Require: Time-series of past resource usage $TS_{\text{usage}}$ of concurrent running tasks, time-series of progress scores $TS_{\text{progress}}$ for a task $tid$, resource $R$, current resource allocation $R_{\text{cur}}$, Estimator scheme (UNIFORM or REGRESSION).
 1: Split $TS_{\text{usage}}$ and $TS_{\text{progress}}$ into equally-sized multiple time-windows, ($W = \{\mathrm{w}_1, \mathrm{w}_2, \dots, \mathrm{w}_\mathrm{t}\}$).
 2: for each $w_\mathrm{i}$ in $W_\mathrm{t}$ do
 3:     if allocation-scheme = UNIFORM then
 4:         Compute the mean ($R_{\text{mean}}$) of the resource usage across all tasks of the same phase and job in the previous time-windows ($w_1 \dots w_{\text{i-1}}$).
 5:     else if allocation-scheme = REGRESSION then
 6:         Get the expected finish time for task $tid$ using the progress-score based model (see Section 4.3.2).
 7:         Compute the expected resource allocation for the next epoch using the linear regression model.
 8:     end if
 9:     return Resource scaling factor, $\alpha$.
10: end for
```

Algorithm 6 Dynamic allocation of resources to tasks (at GRM).

```
Require: Resource type $R$ (CPU or memory), current allocation ($R_{\text{cur}}$), task ID ($tid$), current epoch ($e_{\text{cur}}$), resource scaling factor ($\alpha$).
1: Compute the estimated allocation ($R_{\text{est}}$) using Algorithm 5.
2: if $R_{\text{est}} > R_{\text{cur}}$ then
3:     Dynamically increase (e.g., $\alpha > 1$) the amount of resource allocation $R$ to task $tid$ by a factor ($R_{\text{est}}$ - $R_{\text{cur}}$).
4: else if $R_{\text{est}} < R_{\text{cur}}$ then
5:     Dynamically decrease (e.g., $\alpha < 1$) the amount of resource allocation $R$ to task $tid$ by a factor ($R_{\text{cur}}$ - $R_{\text{est}}$).
6: else
7:     Continue with the current allocation $R_{\text{cur}}$ in epoch $e_{\text{cur}}$.
8: end if
```

is determined by this simple rule – we collect data over as many past time windows till the memory prediction from the resulting estimation model is not worse than the average memory usage across all tasks (*e.g.,* UNIFORM scheme). This serves as a threshold for the maximum number of past time windows to be used for training. From empirical analysis, we found that 10% of the total number of past windows serves as a good estimate for the number of recent past windows. This threshold is adaptive, and depends on the prediction accuracy of the resulting memory usage model. Note that, the parameters of this simple linear model dynamically vary across a task's run-time epochs, and thus not shown here. Algorithm 5 describes the run-time estimated resource computations for a task. Algorithm 6 utilizes Algorithm 5 to perform the on-demand resource allocation.

4.3.2.2 UNIFORM scheme

This scheme is based on the intuitive notion of fair allocation of resources to tasks in order to reduce the run-time resource usage variation, and ensure nearly equal finish times across tasks. Here, the resource entitlement of each map/reduce task is set equal to the *mean* (i.e., average) of the resource allocations across all tasks in the respective map and reduce phase of the same job. However, in practice, due to factors like machine heterogeneity, resource contention and workload imbalance, this scheme might not work well. We are demonstrating this scheme here primarily because of two reasons. First, it is a very simple scheme to implement with negligible performance overheads. Second, it highlights the fact that even a naive but intuitive technique like UNIFORM can achieve reasonable performance benefits.

4.3.3 Implementation Options for *MROrchestrator*

We describe two approaches for the implementation of *MROrchestrator* based on the underlying infrastructure.

4.3.3.1 Implementation on a native Hadoop cluster

Here, we implement *MROrchestrator* on a 24-node Linux cluster, running Hadoop v0.20.203.0. Each node in the cluster has a dual 64-bit, 2.4 GHz AMD Opteron processor, 4GB RAM, and Ultra320 SCSI disk drives, connected with 1-Gigabit Ethernet. Each node runs on bare hardware without any virtualization layer (referred as *native Hadoop*). We use Linux control groups (LXCs) [115] for fine-grained resource allocation in Hadoop. LXCs are Linux kernel-based features for resource isolation, used for similar purposes as in Mesos [72].

4.3.3.2 Implementation on a Virtualized Hadoop Cluster

Motivated by the growing trend of deploying MapReduce applications on virtualized cloud environments [81, 28], we provide the second implementation of *MROrchestrator* on a virtualized Hadoop cluster in order to demonstrate its platform independence, portability and other potential benefits.

We allocate 1 virtual machine (VM) per node on a total of 24 machines (using same above native Hadoop cluster) to create an equivalent 24-node virtualized cluster, running Hadoop v0.20.203.0 on top of Xen [116] hypervisor. Xen offers advanced resource management techniques (like *xm* tool) for dynamic resource management of the overlying VMs. We configure Hadoop to run one task per VM. This establishes a one-to-one equivalence between a task and a VM, giving the flexibility to dynamically control the resource allocation to a VM (using *xm* utility), implying the control of resources at the granularity of individual task.

xm can provide resource isolation, similar to LXCs on native Linux. The core functionalities of GRM, namely Contention Detector and Performance Balancer, are implemented in Dom0. The LRM modules containing the functionalities of Resource Profiler and Estimator are implemented in DomUs. Similar to the native Hadoop case, the TaskTrackers on DomUs collect, profile resource usage data (using Hadoop profiler [114]), and send it to the JobTracker at Dom0 using the Heartbeat messages. We plan to explore the use of LXCs in the virtual cluster, and compare the benefits/trade-offs of these two implementation alternatives as part of our future work.

4.4 Experimental Evaluation

We use a workload suite consisting of the following six representative MapReduce jobs.

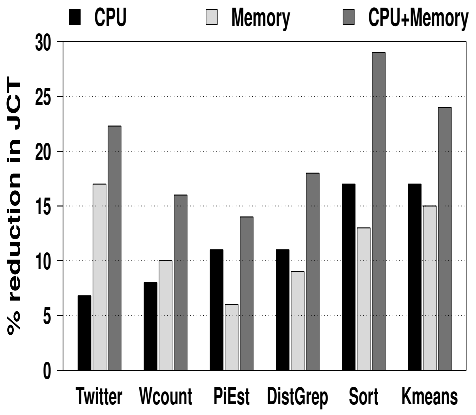
- `Sort`*:* sorts 20 GB of text data generated using Gridmix2 [117] provided random text writer.

- `Wcount`*:* computes the frequencies of words in the 20 GB of Wikipedia text articles [118].
- `PiEst`*:* estimates the value of Pi using quasi-Monte Carlo method that uses 10 million input data points.
- `DistGrep`*:* finds match of randomly chosen regular expressions over 20 GB of Wikipedia text articles [118].
- `Twitter`*:* uses the 25 GB twitter data set [119], and ranks users based on their followers and tweets.
- `Kmeans`*:* constructs clusters in 10 GB worth data points.

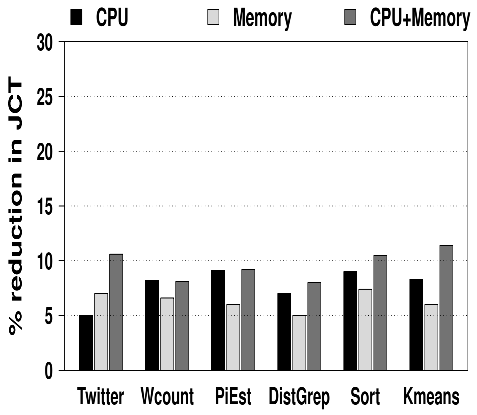
These jobs are chosen based on their popularity and being representative of real MapReduce workloads, with a diverse resource mix. The first four jobs are standard benchmarks available with Hadoop distribution [15], while the last two are in-house MapReduce implementations. The `Sort`, `DistGrep`, `Twitter`, and K-means are primarily CPU and I/O intensive benchmarks, while `Wcount` and `PiEst` are CPU bound jobs. Our primary metrics of interest are reduction in *job completion time* and increase in *resource utilization*. The base case corresponds to the slot-level sharing of resources with Hadoop fair scheduler [69]. Two map/reduce slots per node are configured for the baseline Hadoop.

4.4.1 Results for Native Hadoop Cluster

Figure 4.4(a) shows the percentage reduction in the job completion times (JCTs) (which are in the order of tens of minutes) of the six MapReduce jobs, when each one is run in isolation (*Single job*), with the REGRESSION scheme. *MROrchestrator* is executed in three control modes: (a) *MROrchestrator* controls only CPU (*CPU*); (b) *MROrchestrator* controls only memory (*Memory*); and (c) *MROrchestrator* controls both CPU and memory (*CPU+Memory*) allocations. We show results separately for these three control modes. We summarize (in words, not shown in figures) each result with two values – *average* (mean value across all the 6 jobs) and *maximum* (highest value across all the 6 jobs). We can observe that the magnitude of decrease in JCTs varies at different scales for the three control modes. Across all the 6 jobs, *CPU+Memory* mode tends to yield the maximum reduction in JCTs, while the magnitude of reduction for *CPU* and *Memory* modes varies depending on the job resource usage characteristic. Specifically, we can notice an average and a maximum reduction of 20.5% and 29%, respectively, in the JCTs with *CPU+Memory* mode of *MROrchestrator*. Further, we see that `Sort` job makes extensive use of CPU (map phase), memory and I/O (reduce phase), and benefits the most. The percentage reduction in the JCTs for other five jobs varies depending on their resource sensitiveness. It is to be noted that larger jobs (with respect to both bigger input size and longer finish time) like `Twitter`, `Sort` and `Kmeans` tend to benefit more with *MROrchestrator*, when compared to other relatively shorter jobs (`PiEst`, `DistGrep`, `Wcount`). The reason being that larger jobs run in multiple map/reduce phases, and thus can benefit more from the higher utilization achieved by *MROrchestrator*.



(a) REGRESSION scheme

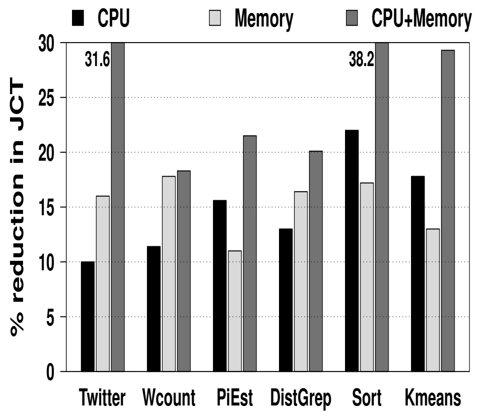


(b) UNIFORM scheme

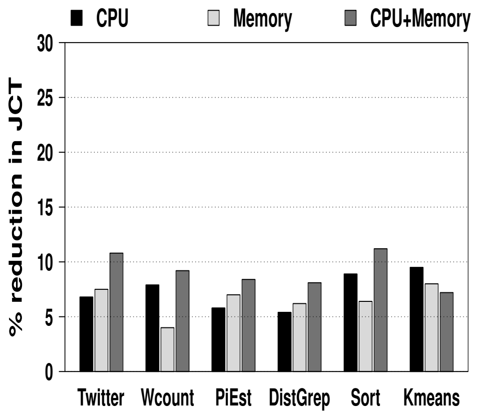
Figure 4.4. Reduction in Job Completion Time (JCT) for *Single job* case in native Hadoop cluster.

We next analyze the performance of *MROrchestrator* with the UNIFORM scheme. Figure 4.4(b) shows the results. We can observe an average and a maximum reduction of 9.6% and 11.4%, respectively, in the completion times. As discussed in Section 4.3.2.2, UNIFORM is a simple but naive scheme to allocate resources. However, it achieves reasonable performance improvements, suggesting the generality of *MROrchestrator*.

Next, we analyze the completion times of individual jobs in the presence of other concurrent jobs (*Multiple jobs*). For this, we run all the six jobs together.



(a) (a) REGRESSION scheme

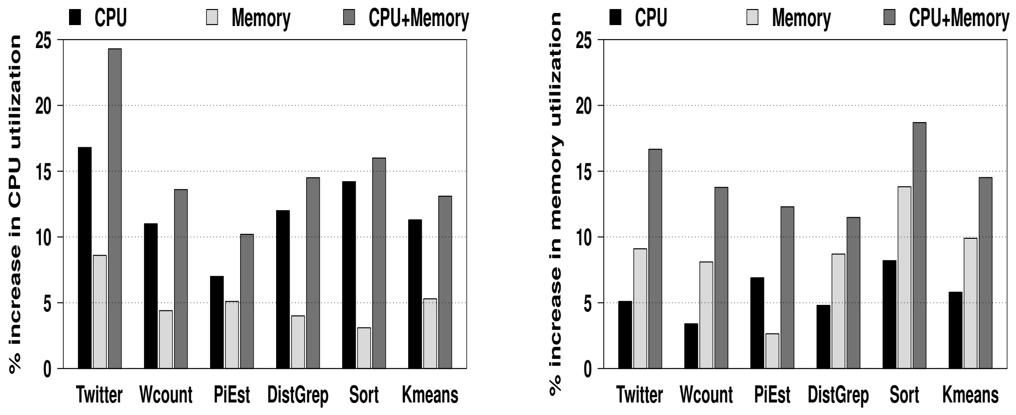


(b) UNIFORM scheme

Figure 4.5. Reduction in Job Completion Time (JCT) for *Multiple jobs* case in native Hadoop cluster.

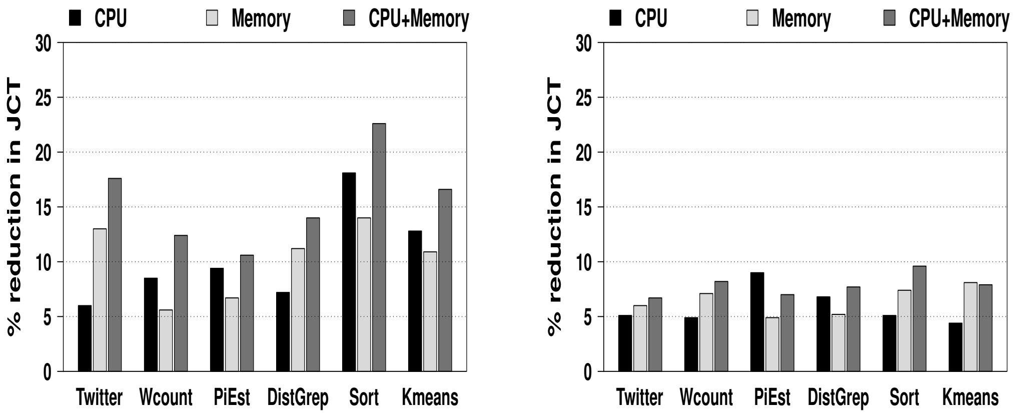
Figure 4.5(a) shows the percentage reduction in the completion times of the 6 jobs with the REGRESSION scheme. We observe an average and a maximum reduction of 26.5% and 38.2%, respectively, in the finish times with *CPU+Memory* mode. With the UNIFORM scheme and *CPU+Memory* (Figure 4.5(b)), we observe 9.2% (average) and 12.1% (maximum) reductions in JCTs.

Figures 4.6(a) and 4.6(b) show the percentage increase in the CPU and memory utilization for *Multiple jobs* scenario. With the REGRESSION scheme and *CPU+Memory*, we see an increase of 15.2% (average) and 24.3% (maximum) for CPU; 14.5% (average) and 18.7% (maximum) for memory. We observe an average increase of 7% and 8.5% in CPU and memory utilization, respectively for the *Single job* case with *CPU+Memory* mode. With the UNIFORM scheme, the percentage increase seen in CPU and memory is within 10% both for *Single job* and *Multiple jobs* cases. Note that first, the benefits seen with *MROrchestrator* are higher for environments with *Multiple jobs*. This is due to better isolation, dynamic allocation and global coordination provided by *MROrchestrator*. Second, the control of both CPU and memory (*CPU+Memory*) with *MROrchestrator* yields the maximum benefits.



(a) CPU improvement (b) Memory improvement

Figure 4.6. Improvement in CPU and memory utilization in native Hadoop cluster with *Multiple jobs*.

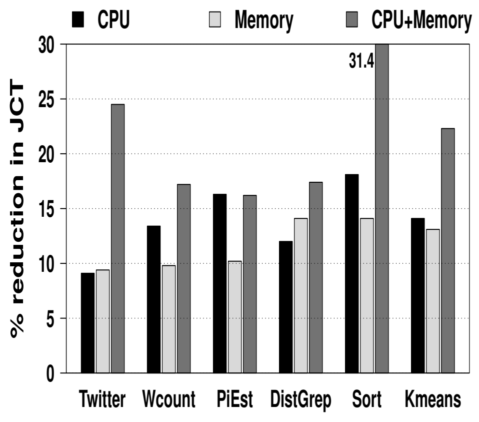


(a) REGRESSION scheme (b) UNIFORM scheme

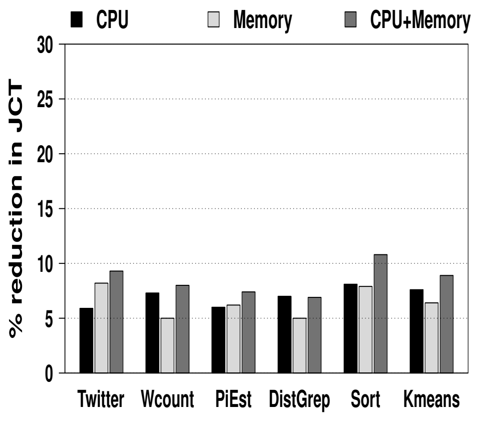
Figure 4.7. Reduction in Job Completion Time (JCT) for *Single job* case in virtualized Hadoop cluster.

4.4.2 Results for Virtualized Hadoop Cluster

Figure 4.7(a) and Figure 4.7(b) show the percentage reduction in job completions for the *Single job* case with REGRESSION and UNIFORM schemes, respectively. We can notice a reduction of 15.6% (average) and 22.6% (maximum) with the REGRESSION scheme and *CPU+Memory* mode. With the UNIFORM scheme, the corresponding reduction is 7.8% (average) and 9.6% (maximum). When all the jobs are run concurrently, the percentage reduction is 21.5% (average) and 31.4% (maximum) for the REGRESSION scheme (Figure 4.8(a)). With the UNIFORM

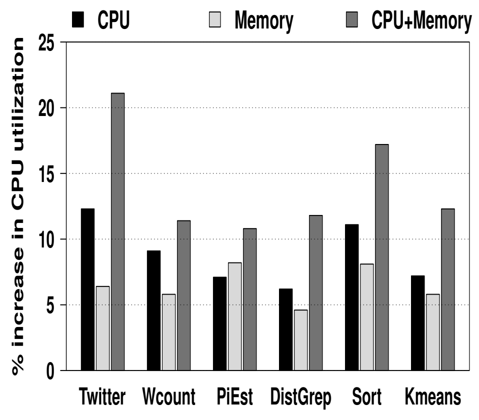


(a) (a) REGRESSION scheme

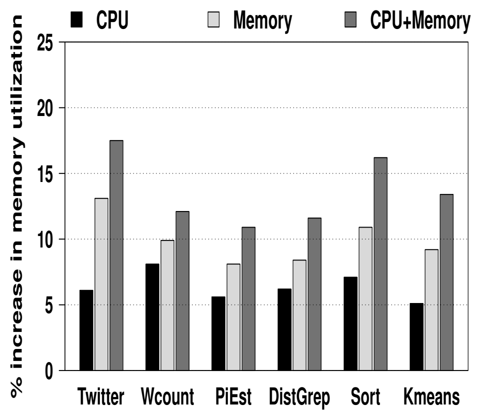


(b) UNIFORM scheme

Figure 4.8. Reduction in Job Completion Time (JCT) for *Multiple jobs* case in virtualized Hadoop cluster.



(a) CPU usage improvement



(b) Memory usage improvement

Figure 4.9. Improvement in CPU and memory utilization in virtualized Hadoop cluster with *Multiple jobs*.

scheme (Figure 4.8(b)), the percentage decrease in finish time is 8.5% (average) and 10.8% (maximum), respectively.

The resource utilization improvements are plotted in Figure 4.9(a) and Figure 4.9(b). We observe an increase in the utilization of 14.1% (average) and 21.1% (maximum) for CPU (Figure 4.9(a)); increase of 13.1% (average) and 17.5% (maximum) for memory (Figure 4.9(b)). These numbers correspond to REGRESSION scheme with *CPU+Memory*. For UNIFORM scheme, the percentage increase in CPU and memory utilization is less than 10%.

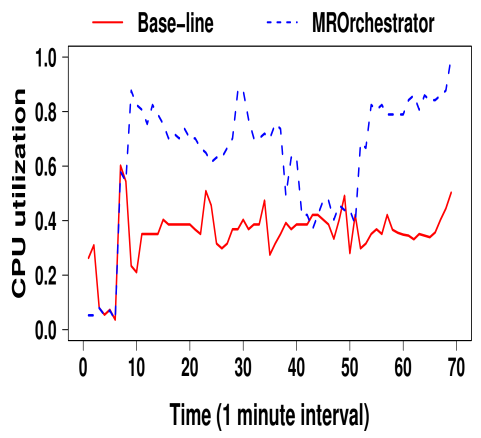
The performance benefits obtained from the implementation of *MROrchestrator* on a native Hadoop is marginally better than corresponding implementation on a virtualized Hadoop. There are two possible reasons for this. First, it is due to the CPU, memory or I/O (in particular) performance overheads associated with the Xen virtualization [120]. Second, configuring one task per virtual machine to achieve one-to-one correspondence between them seems to inhibit the degree of parallelization. However, the difference is not much, and we believe with the growing popularity of MapReduce in virtualized cloud environments [29], coupled with advancements in virtualization, the difference would shrink in near future.

To illustrate the dynamics of *MROrchestrator*, we plot Figure 4.10(a) and Figure 4.10(b), which show the snapshots of CPU and memory utilization of the native Hadoop cluster with and without *MROrchestrator* (*base-line*), and running the same workload (all 6 MapReduce jobs running concurrently). We can observe that *MROrchestrator* provides higher utilization compared to the *base-line*, since it is able to provide better resource allocation aligned with task resource demands.

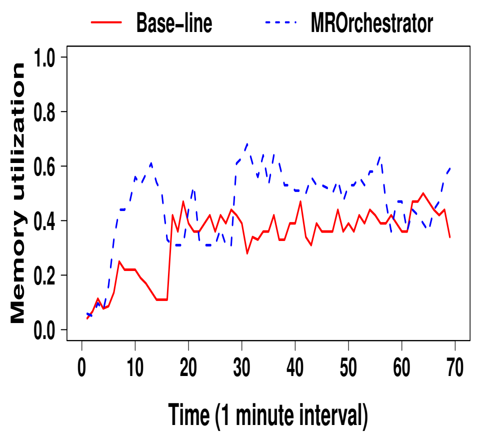
4.4.3 *MROrchestrator* with Mesos and NGM

There are two contemporary resource scheduling managers – Mesos [72] and Next Generation MapReduce (NGM) [113] that provide better resource management in Hadoop. We believe *MROrchestrator* is complementary to both Mesos and NGM, and thus, integrate them to derive added benefits.

We first separately compare the performance of Mesos, NGM and *MROrchestrator*, normalized over the base case of default Hadoop with fair scheduling. Figure 4.11(a) shows the results. We can notice that the performance of *MROrchestrator* is better than both Mesos and NGM for all but one job (`PiEst`) (possibly because `PiEst` is a relatively shorter job, operating mostly in single map/reduce phase). The average (across all the 6 jobs) percentage reduction in JCT observed with *MROrchestrator* is 17.5% and 8.4% higher than the corresponding reduction seen with Mesos and NGM, respectively. We can observe that NGM has better performance than Mesos with the DRF scheme (a

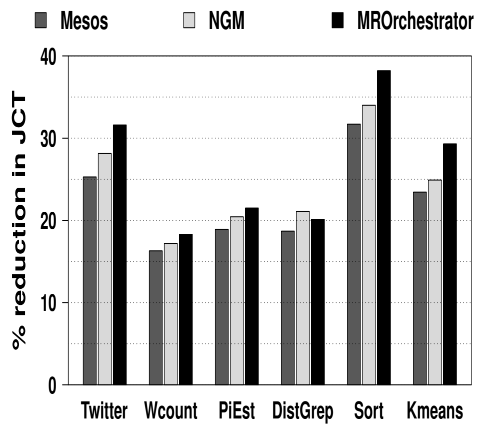


(a) CPU utilization time-series

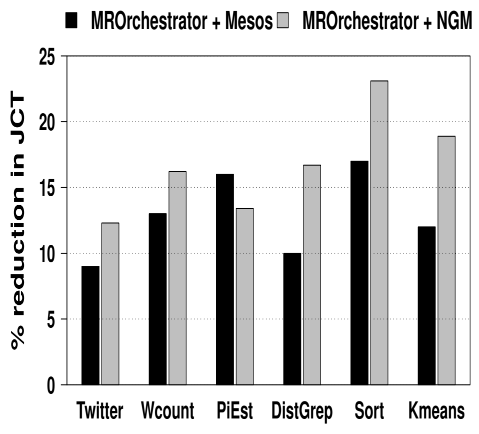


(b) Memory utilization time-series

Figure 4.10. Illustration of the dynamics of *MROrchestrator* in improving the cluster resource utilization.



(a) *MROrchestrator*, Mesos, NGM



(b) *MROrchestrator* + Mesos + NGM

Figure 4.11. Illustration of the performance benefits of the integration of *MROrchestrator* with Mesos and NGM, respectively.

multi-resource scheduling scheme recently proposed for Mesos in [17]). One possible reason is the replacement of slot with the resource container unit in NGM, which provides more flexibility and finer granularity in resource allocations [113].

We next demonstrate the benefits from the integration of *MROrchestrator* with Mesos (*MROrchestrator + Mesos*) and NGM (*MROrchestrator + NGM*), respectively. The results are shown in Figure 4.11(b). We observe an average and a maximum reduction of 12.8% and 17% in JCTs for *MROrchestrator + Mesos*.

For *MROrchestrator + NGM*, the average and maximum decrease in JCTs is around 16.6% and 23.1%, respectively. Further, we observe an increase of 11.7% and 8.2% in CPU and memory utilization, respectively for *MROrchestrator + Mesos*. For *MROrchestrator + NGM*, the corresponding increase in CPU and memory utilization is around 19.2% and 13.1%, respectively.

There are two main reasons for the observed better performance with *MROrchestrator*'s integration. First, irrespective of the allocation units, the static characteristics in Mesos and NGM still persist. On the other hand, *MROrchestrator* dynamically controls and provides on-demand allocation of resources based on the run-time resource profiles. Second, the inefficiencies that arise due to the lack of global coordination among nodes have yet not been addressed both in Mesos and NGM. *MROrchestrator* incorporates such global coordination.

4.4.4 Performance Overheads of *MROrchestrator*

There are some important design choices concerning the performance overheads of *MROrchestrator*. First, the frequency (or epoch duration) at which LRMs communicate with GRM is an important performance parameter. We performed detailed sensitivity analysis to determine the optimal frequency value by taking into consideration the trade-offs between prediction accuracy and performance overheads due to message exchanges. Based on the analysis, *20 seconds* (four times the default Hadoop heartbeat message frequency) is chosen as the epoch duration. Second, we discuss the expected delay in detecting and resolving resource bottlenecks in *MROrchestrator*. It consists of four major parts (overhead for each is with respect to the 20 seconds epoch duration): (i) resource usage measurements are collected and profiled every 5 seconds (equals Heartbeat message interval). The associated delay is negligible because of the use of a light-weight, built-in Hadoop profiler. (ii) time taken in resource bottleneck detection by Contention Detector is of the order of 10s of milliseconds; (iii) time overhead to build the predictive models by Estimator is less than 1% and 10% for UNIFORM and REGRESSION schemes, respectively; and (iv) time overhead to resolve contention by Performance Builder is within 4%.

Model Accuracy: With the REGRESSION scheme, 90% of predictions are within 10% and 19% accuracy for CPU and memory usage models, respectively. For UNIFORM scheme, the percentage error in resource estimation is within 35%.

Scalability: MROrchestrator can scale well with larger systems since the core modules (Estimator and Performance Builder) work on a fixed amount of recent history data, and thus, are largely independent of the increase in scale.

4.5 Chapter Summary

In this chapter, we analyzed the disadvantages of fixed-size and static slot-based resource allocation in Hadoop MapReduce. Based on the insights, we proposed the design and implementation of a flexible, fine-grained, dynamic and coordinated resource management framework, called *MROrchestrator*, that can efficiently manage the cluster resources. Results from the implementations of *MROrchestrator* on two 24-node physical and virtualized Hadoop clusters, with representative workload suites, demonstrate that up to 38% reduction in job completion times, and up to 25% increase in resource utilization can be achieved. We further show how contemporary resource scheduling managers like Mesos and NGM, when augmented with *MROrchestrator* can boost their performance.

Chapter 5

Hierarchical MapReduce Scheduler for Hybrid Data Centers

5.1 Introduction

Virtualization has evolved as a key technology to support agile and dynamic Information Technology (IT) infrastructure, which forms the base of large distributed systems like data centers and clouds. Virtualization enables autonomic management of underlying hardware, server sprawl reduction through workload consolidation, and dynamic resource allocations for better throughput and energy efficiency. Consequently, major cloud providers like Amazon EC2, RackSpace and Microsoft Azure utilize server virtualization to efficiently share resources among customers, and allow for rapid elasticity.

Despite these numerous benefits, virtualization introduces an extra software layer to the system stack, incurring overheads to native performance. Analyses with generic benchmarks have shown the virtualization overheads to be around 5% for computation and 15% for I/O workloads [121]. While these virtualization overheads have continued to fall with the introduction of virtualization-aware hardware [91], the effects are still large enough that many companies like Google and Facebook still prefer physical machines (PMs) over virtual machines (VMs) to run their core applications like Web search [122]. For data analytics frameworks such as Hadoop MapReduce [15] that allow for efficient large scale distributed computation over massive data sets [14], virtualized cloud platforms seem like a natural fit for providing the desired elastic scalability. However, virtualization in clouds is also known to incur performance overheads, particularly when used for I/O intensive MapReduce activities [83, 81]. As a result, MapReduce users are often left with the choice of either maximizing performance with a native cluster or obtaining ease of use and resource efficiency with a virtualized environment.

Today's data centers offer two different modes of computing platforms - native clusters and virtual clusters. Both these environments have their own strengths and weaknesses. For example, a native cluster is better for batch workloads like MapReduce from the performance standpoint, but usually suffers from poor utilization, and high hardware and power cost. A virtual cluster, on the other hand, is attractive for transactional workloads from consolidation and cost standpoints, but may not provide competitive performance like a native cluster. Intuitively, a hybrid compute platform consisting of a virtualized as well as a native cluster, should be able to exploit the benefits of both environments for providing a better cost-effective infrastructure. In this chapter, we explore this design alternative, which we call *Hybrid data center*, and demonstrate its advantages for supporting both interactive and batch workloads.

Transactional applications like interactive web services and virtual desktop environments are prime candidates for virtualization. For supporting the SLA requirements of interactive applications, resources are generally over-provisioned, leading to poor utilization [84]. To exploit the potentials of both native and virtual cluster, we leverage the over-provisioning of bursty interactive applications by intelligently consolidating batch MapReduce jobs using the spare resources available on a virtualized platform. This allows for reaping the benefits of high consolidation in multi-tenant systems. This hybrid infrastructure presents different trade-offs across various design metrics like performance, cost, energy and resource utilization between native, virtual and hybrid design choices. For example, a native cluster incurs higher cost in terms of power and physical hardware procurement, but lowers SLA violations, whereas a virtualized cluster achieves higher resource utilization, elasticity and lower energy cost, but relatively more SLA violations. Hybrid data center achieves a better balance between all these design criteria, making it a desirable cluster configuration option.

For facilitating such a hybrid compute infrastructure, this chapter presents the design and implementation of a 2-phase hierarchical scheduler, called *HybridMR*, that judiciously allocates virtual and physical resources to applications. Contrary to the traditional workload placement schemes that completely isolate batch MapReduce and interactive workloads, *HybridMR* consolidates the workload mix in a heterogeneous infrastructure to achieve higher performance, utilization and energy efficiency targets. The design of such a scheduler requires the knowledge of how different MapReduce jobs are impacted by virtualization overheads, estimates of their resource needs, and an understanding of how batch jobs will impact the performance of interactive jobs co-hosted in VMs on the same host. *HybridMR* addresses these challenges through its 2-phase scheduler design. Its first phase places MapReduce jobs on physical or virtual nodes depending on the expected virtualization overheads. Once a set of MapReduce jobs have been selected to run on the virtual cluster, collocated with interactive applications, the second phase scheduler then decides *how much* resources can be safely assigned to them. For this, it makes use of developed predictive models for understanding the run-time resource interference between the interactive and batch jobs, and employ dynamic resource management techniques to provide the best effort delivery to MapReduce jobs, while upholding the SLAs of interactive applications. *HybridMR* targets improving institutional and enterprise intranet environments, where the data centers run in a hybrid fashion, comprising of evolving cluster mix of native and virtual nodes.

In summary, this research segment makes the following contributions:

- This chapter makes a case for hybrid data centers with native and virtual machines for co-hosting both transactional and batch workloads for exploiting the best of these two computing worlds – native performance with virtualization benefits.

- Through detailed analysis, this chapter investigates the trade-offs and benefits of running Hadoop in a virtualized platform, and leverage the insights in the design of *HybridMR*.
- We have designed and implemented *HybridMR*, a 2-phase hierarchical scheduler, for the effective placement of MapReduce jobs in a virtualized platform, while upholding the SLAs of interactive applications, through further dynamic resource management.
- Evaluations on a hybrid cluster consisting of 24 physical servers, and 48 virtual machines, with diverse workload mix of transactional and MapReduce applications, demonstrate that *HybridMR* can achieve up to 40% improvement in job completion times over virtual Hadoop, 45% improvement in resource utilization, and 43% savings in energy, both relative to native Hadoop. Further, this chapter demonstrates the possibility to dynamically vary the native and virtual cluster configurations to accommodate variations in workload mix for maximizing the performance and energy benefits. These results suggest that a hybrid data center configuration could be a better cost-effective solution than either-or native and virtual modes of computation.

The rest of this chapter is organized as follows. Section 5.2 presents the underlying motivation of the problem addressed. Section 5.3 describes the design and implementation of *HybridMR*. The evaluations are presented in Section 5.4, followed by the chapter summary in Section 5.5.

5.2 Motivation

5.2.1 MapReduce in Virtual Environment

HybridMR targets leveraging the unused spare resources in virtual clusters running interactive applications by consolidating batch MapReduce jobs. In this context, we address the following questions that are critical for running batch workloads like Hadoop MapReduce on a hybrid cluster.

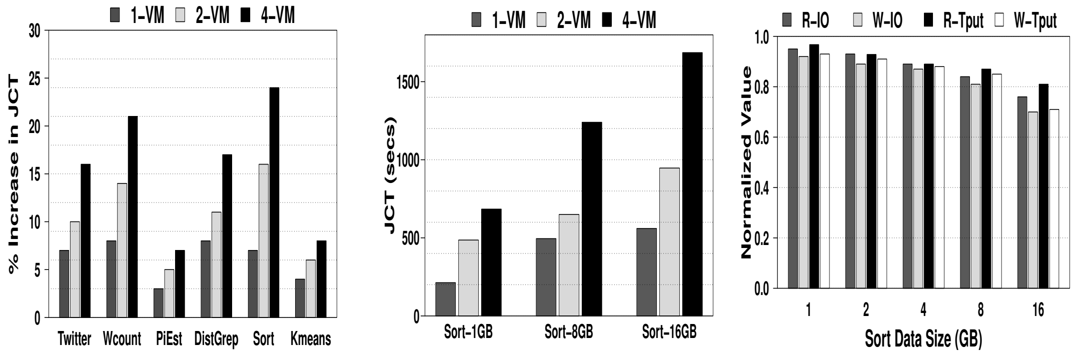
Q1. What are the challenges, benefits and trade-offs of running Hadoop on a virtual versus equivalent native cluster?

Q2. How does the 'data sticky-ness' of Hadoop affect the overall system performance in virtual Hadoop?

Q3. What system and application level changes are required to realize Hadoop's deployment on hybrid data centers?

First, we perform detailed analysis and present empirical evidences demonstrating various challenges, benefits and trade-offs involved in deploying Hadoop MapReduce in a virtual cluster. We also highlight the various key design choices that impact Hadoop virtualization. Second, we present the split architecture of Hadoop that addresses the movement of large data across a virtual cluster during live VM migration. For this study, each VM is configured with 1 vCPU and 1024 MB Memory, and runs on dual-core, 4 GB RAM servers. We use a total of 1-24 PMs and 1-48 VMs, depending on the requirements of each experiment. Further details about the experimental platform, including the MapReduce benchmarks used are described in Section 5.3.

We start with comparing the performance of MapReduce benchmarks on native versus virtual environments. In Figure 5.1(a), the Y-axis represents the percentage increase in job completion time (JCT), with respect to a physical cluster. We can observe that with the increase in the number of VMs per PM, the performance (as quantified by JCT) of I/O bound jobs like `Twitter`, `Wcount`, `DistGrep`, `Sort` in virtual cluster is 7-24% worse than the physical cluster. For CPU bound jobs like `PiEst`, `Kmeans`, the performance difference is less than 8%. Moreover, for all benchmarks, as the size of input data increases, the performance gap widens, as evident in Figure 5.1(b) (only `Sort` shown). To investigate this behavior, we benchmark the performance of Hadoop Distributed File System (HDFS) in terms of read and write I/O, read and write throughput on a virtual cluster normalized with respect to the corresponding native cluster. We use the Hadoop TestDFSIO [15] benchmark, and the results are shown in Figure 5.1(c).

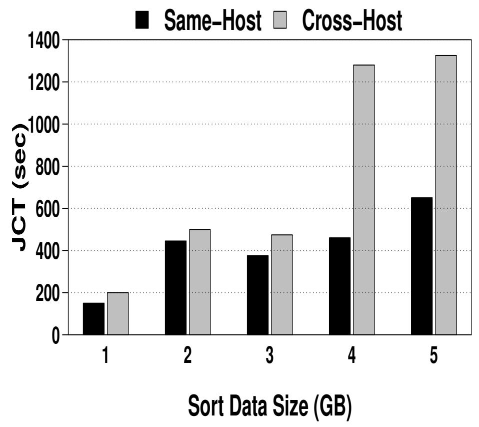


(a) MapReduce performance (b) Data size impact (c) HDFS performance

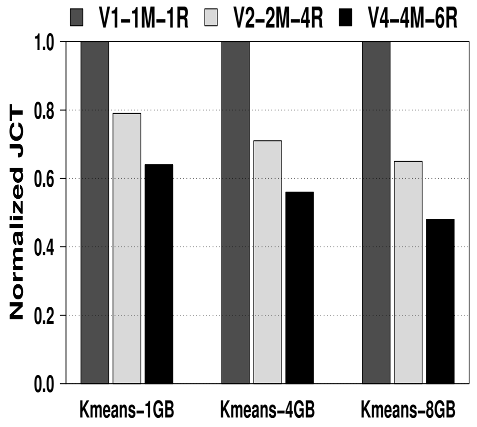
Figure 5.1. Illustration of the virtualization overheads on Hadoop performance. Y-axis in (a), (c) normalized with respect to equivalent physical cluster. Figure 5.1(c): R/W-IO: Read/Write IO in MB/sec (Avg. IO rate); R/W-Tput: Read/Write-Throughput in MB/sec (total number of bytes/sum of processing times). In these experiments, total 48 VMs are used.

We notice that the virtual cluster performs relatively worse than the native cluster, and again this performance gap increases with the increase in data size. The reasons for this poor performance are: (i) increase in contention for shared I/O resources across multiple VMs when large amount of data is transferred between map and reduce phase; (ii) poor performance of HDFS when multiple VMs concurrently read/write data blocks from/to HDFS; and (iii) presence of more stragglers, hence more speculative tasks, causing inefficient resource utilization [14].

In another experiment, we study the impact of Hadoop VMs crossing multiple hosts due to large size of virtual cluster, limited host resources, and scattered data. For this, we create a 16 VMs virtual Hadoop cluster. In *Cross-Host*, the 16 VMs are equally distributed across 8 dual-core PMs, while in *Same-Host*, all the 16 VMs are equally consolidated on 2 PMs. Figure 5.2(a) shows the JCT of `Sort` in these two cases. The poor performance of *Cross-Host* compared to Same-Host is due to increase in the network I/O delay caused by remote VMs communication in *Cross-Host*. Further, the JCTs in each case increase as the input data scales up. An interesting observation here is that despite the *Cross-Host* having access to more cores (each VM gets 1 vCPU) when compared to *Same-Host* (each VM gets 0.25 vCPU), the performance of the former is still worse than the latter. Thus, when the data size and the number of concurrent running map/reduce tasks increase,

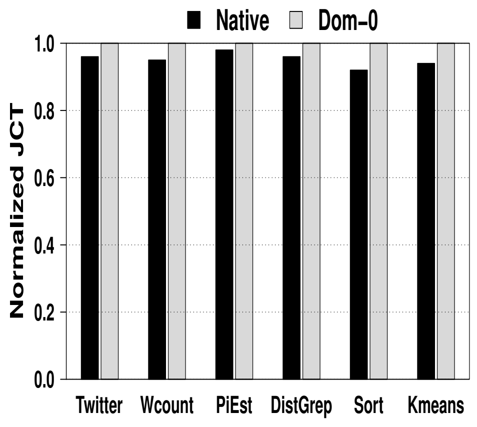


(a) Network I/O effect

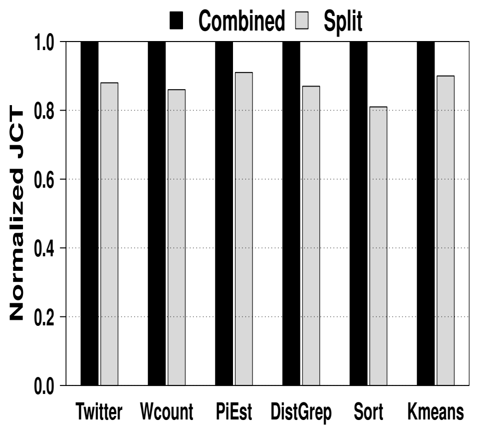


(b) More CPU cycles effect

Figure 5.2. In (a), 16 VMs are used; 48 VMs are used in (b). In (b): V1, V2, V4 denote 1, 2, 4 VMs per PM. M and R denote number of Map and Reduce tasks. Y-axis in (b) is normalized with respect to Native.



(a) Native Vs. Dom-0



(b) Hadoop *Split* architecture

Figure 5.3. In (a), 48 VMs are used. Y-axis in (a) is normalized with respect to Native. Y-axis in (b) is normalized with respect to Combined.

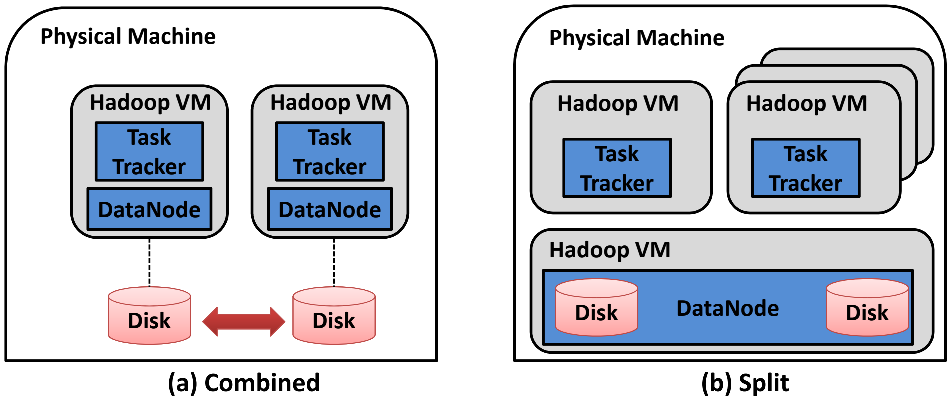
the network communication overheads become the main resource bottleneck, and decides the overall performance.

Hadoop MapReduce can also perform better in virtual environment in certain scenarios. For example, CPU-bound jobs like `Kmeans` can benefit from having more VMs available, as shown in Figure 5.2(b). Further, this performance increase is higher for larger data size. This is because, a CPU-bound job can leverage more number of map/reduce tasks to finish fast due to the availability of more compute cycles, and hence more slots with more VMs on multi-core hosts. Note that in Figure 5.1(a), CPU-bound `Kmeans` tends to perform worse with more number of VMs per PM. However, this is opposite to the result shown in Figure 5.2(b). This is because, for Figure 5.2(b) scenario, more number of map and reduce tasks can be launched to exploit the free CPU cycles (*i.e.,* slots), compared to the scenario in Figure 5.1(a), where fewer number (default 2) of concurrent mappers and reducers are present.

We next explore leveraging Xen's split architecture to run Hadoop in the privileged domain (Dom-0). Results in Figure 5.3(a) show that Dom-0 provides near native performance, with overheads less than 5%. This opens the possibility of a "flexibly virtualized" cluster, where some machines can be easily transitioned from running VMs to quasi-native applications in Dom-0 [91]. This supports hybrid clusters, whose configuration can be adjusted flexibly and on-demand.

Finally, one of the other concerns of deploying Hadoop on virtualized environment is the challenge to deal with the inherent 'data sticky-ness' of Hadoop, *i.e.,* how to deal with the movement of large amount of data living on Hadoop VMs, which may lead to high inefficiency both in terms of time and network communication overheads, thus defeating many of the benefits of virtualization. This also affects elasticity, *i.e.,* adding/removing Hadoop VMs to increase/reduce compute resources is ineffective due to the tight coupling between the compute and data storage layer. A prospective workaround to this problem is to place the TaskTracker (compute nodes) and DataNode (storage node) on separate VMs [82]. This *split* architecture (see Figure 5.4) maintains data locality, and removes the constraint of moving large amounts of data around the cluster during VM migrations, since the data stays resident in the HDFS data layer, while the amount of map/reduce (compute) VMs can vary as required. We did some initial evaluations to assess the performance implications of this *split* architecture. Figure 5.3(b) shows the results. We observe an average of 12.8% improvement in JCT over the default Hadoop (*Combined*). We adopt the *split* architecture in this research study.

To summarize, the above analyses clearly suggest that the performance of virtualized Hadoop is dependent on and governed by a multitude of design



Compute and storage layer are combined on a single VM in (a), while they are split across separate VMs in (b). Efficient virtual networking between VMs in a host allows (b) to maintain compute-data locality.

Figure 5.4. Hadoop Split Architecture.

factors, and thus coming up with an optimal configuration both at the job and cluster level is non-trivial, making scheduling and resource management difficult. Some of the key observations are (i) Hadoop performance on a virtual cluster is relatively worse than the equivalent physical cluster. The magnitude of performance degradation varies depending on the nature of the job. For example, jobs which are I/O and network intensive like `Sort` suffer more performance degradation than CPU-bound jobs like `PiEst`; (ii) input data size directly affects the magnitude of performance difference; (iii) Hadoop's performance difference between native and Dom-0 is marginal; (iv) *Split* Hadoop architecture is an optimized deployment on a virtual environment.

5.3 Design of *HybridMR*

This section describes the overall architecture of *HybridMR* (as shown in Figure 5.5), which operates in two phases. In the first phase, *HybridMR* attempts to classify incoming MapReduce jobs based on the expected virtualization overheads, and uses that information to automatically guide placement between physical and virtual machines. The second phase performs dynamic resource management to minimize interference and improve performance of collocated interactive and batch jobs. Specific details of these phases are described below.

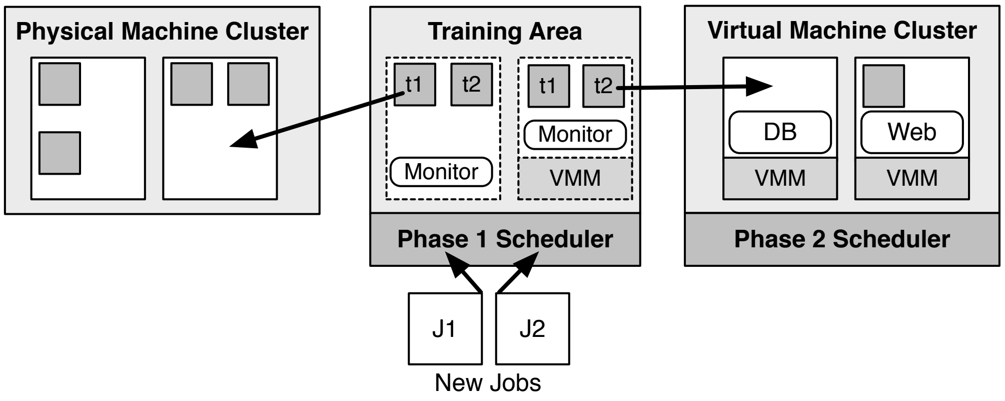


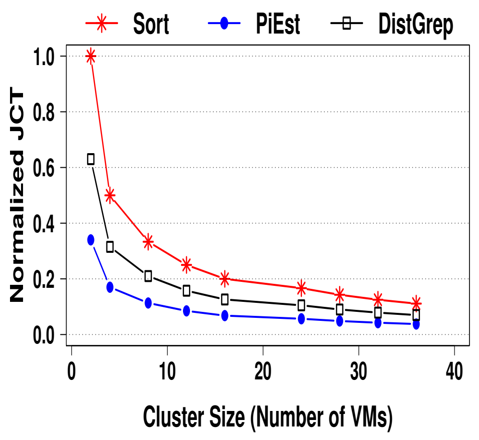
Figure 5.5. Overview of *HybridMR*.

5.3.1 Phase I Scheduler

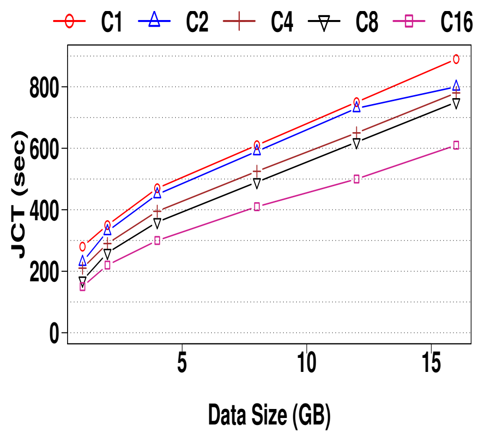
The main goal of this phase is to differentiate between workloads that should be scheduled on PMs or VMs running in a hybrid cluster. Since, our objective is to harness the spare resources on VMs running interactive applications, the interactive applications by default are assigned to the virtual cluster on their arrival, and the placement of the MapReduce jobs is governed by this phase. Thus, when a MapReduce job arrives, it is initially started separately on a small training cluster containing both physical and virtual environments, respectively (see Figure 5.5). We make use of statistical profiling techniques to estimate the JCTs of MapReduce jobs (see Section 5.3.1.1). By comparing the estimated JCTs of the two instances of the job, corresponding to its run on native and virtualized servers, the level of performance overhead (quantified by JCT) incurred by the virtualization layer is estimated. If the overhead is not significant, then the job is selected for deployment on the virtual cluster, else it is run on a separate physical cluster. To estimate JCT, we leverage a job profiling technique, whose details are described next.

5.3.1.1 Job Profiling

We profile MapReduce jobs to estimate their JCTs prior to execution. MapReduce jobs are data-intensive and massively parallel, hence their JCTs are predominantly



(a) Impact of cluster size



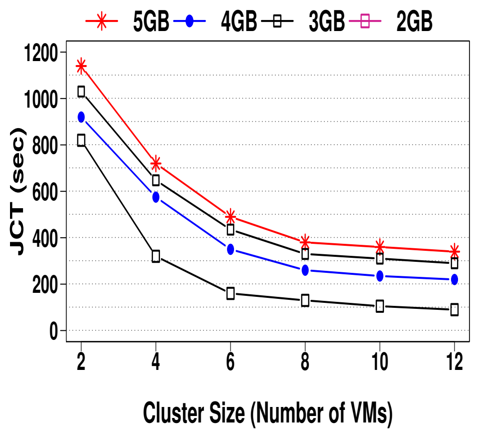
(b) Impact of input data size

Figure 5.6. Dependence of job completion time on cluster size (end-to-end), and input data size. In (b), C1, C2, C4, C8, C16 represent virtual cluster with 1, 2, 4, 8, 16 nodes, respectively. `Sort` is used in (b).

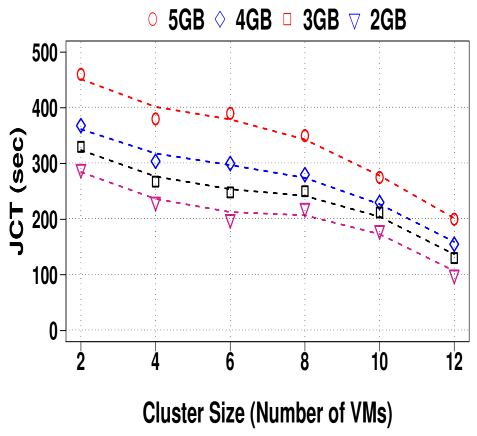
dependent on two factors: (i) input data set size; and (ii) resource set size, *i.e.,* number of nodes in the cluster. The estimation scheme accumulates a database of past execution history of jobs (in terms of job completion times, corresponding to different input data set sizes and cluster sizes). During training, a job is run on a representative small cluster and with a smaller data set, and extrapolation techniques are used to estimate the run-time if the job were run on the full cluster and data set. The profile database maintains separate run-times for map and reduce phases to account for the differences across phases. The exact association of a job's profile with the cluster size and input size is described next.

Cluster size: To quantify the dependence of cluster size on job completion time, we measure the total time as well as the time taken by its map and reduce phases separately, against different cluster sizes. In Figure 5.6(a), we observe that for a given data size, the JCT of a job follows an inverse relation to the cluster size. Similar inverse relation follows for the completion time of map phase as shown in Figure 5.7(a). However, the reduce phase completion time follows a piece-wise non-linear relation with the cluster size as shown in Figure 5.7(b). This is due to the fact that the reduce phase's dependence on data size is highly erratic.

Data size: For a given cluster size, the JCT is linearly proportional to the input data size, as demonstrated in Figure 5.6(b). This indicates that by simple



(a) Map Phase



(b) Reduce Phase

Figure 5.7. Dependence of job completion time on separate map and reduce phase. `Sort` is used in (b) and (c).

linear extrapolation, we can accurately estimate the JCT of a new job instance with a different data size. However, when sufficient historical job profiles are not available, we can still profile on a subset of data, and then use linear regression based extrapolation to get the JCT for the actual input size.

Thus, by building the job profiles using different data sizes and/or cluster sizes, this module estimates the job completion time. The end-to-end mechanism for job profiling based on the above logic is outlined in Algorithm 7. To elaborate further, for a newly arriving MapReduce job, this algorithm searches the database for matching cluster size and/or data size. On a match, the corresponding profiled job completion time estimate is retrieved. When there is no match, appropriate extrapolation technique based on map and reduce phase, is used to derive the estimated job completion time.

We build our database of job profiles using the average across three runs of each job instance. Note that a similar procedure for profiling MapReduce jobs has been explored in previous studies [78, 67]. There are also other more complex profiling techniques [67, 123] for MapReduce jobs, where profiling is done for fine-grained stages of MapReduce framework. Also, the technique can be extended for online profiling [67, 78]. However, we observed that this simple profiling technique works best in practice for our purpose, incurring low overheads, while producing reasonable prediction accuracy. For example, Figure 5.8(a) shows the actual JCTs

Algorithm 7 MapReduce Job Profiling Algorithm.

```
Require: $Q$: queue of incoming jobs, each specifying the input data size and/or cluster
    size; $DB_{profile}$: profile database, containing historic observations of job completion
    times (JCTs) (end-to-end and separate for map and reduce phases), along with
    corresponding cluster sizes and input data set sizes.
 1: for each job $J_i$ in $Q$=$J_1$,$J_2$,...,$J_n$ do
 2:   if LOOKUP_CLUSTERSIZE($DB_{profile}, J_i$) $\neq$ $NULL$ &
      LOOKUP_DATASIZE($DB_{profile}, J_i$) $\neq NULL$ then
 3:     $JCTestimated$=Retrieve($DB_{profile}$, $J_{i(csize)}$, $J_{i(dsize)}$)
 4:   else if $DB_{profile}$ does not contain exact match for $J_i$'s input configuration then
 5:     if $DB_{profile}$ contains different data size values for the same cluster size (see
        Figure 5.6 (b)) then
 6:       Do linear extrapolation to get $JCT_{estimated}$
 7:     else if $DB_{profile}$ contains different cluster size values for the same data size
        (see Figure 5.6 (a)) then
 8:       Do separate Map and Reduce phase based extrapolation to get $JCT_{estimated}$
          (see Figures 5.7 (a) and (b))
 9:     end if
10:   end if
11:   return $JCT_{estimated}$
12: end for
```

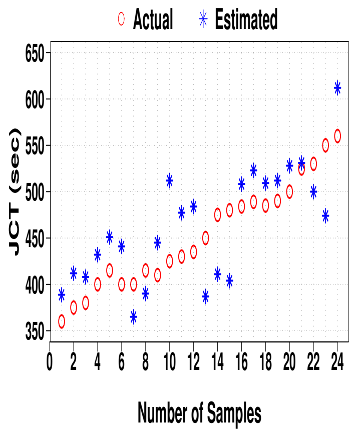
and estimated JCTs obtained from profiling of `Sort` on 10 GB. We got similar results for the other MapReduce benchmarks, and our profiling scheme results in an average error of 10.8% with standard deviation of 9.7%.

5.3.1.2 Placement of MapReduce Jobs

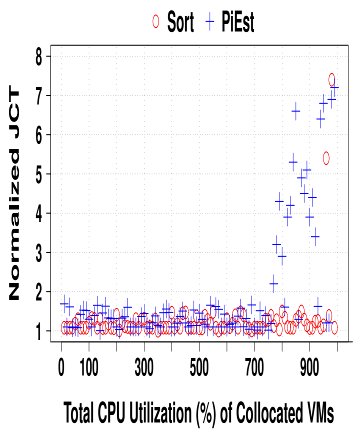
When a job is submitted to the system, depending on the type of the job (transactional or batch), and its desired completion time, the heuristic outlined in Algorithm 8 determines its initial placement on the native or virtual nodes. Note that, Phase I simply steers the initial placement of the job, the exact configuration of VMs or PMs, where the job will be run during its lifetime is determined by Phase II scheduler, as described next.

Algorithm 8 Job Placement Algorithm.

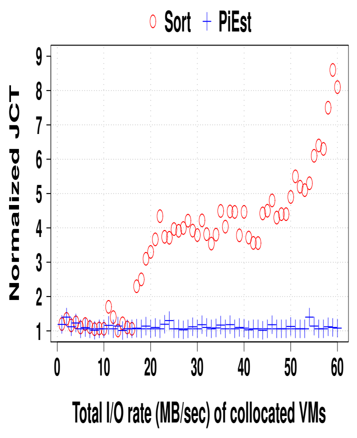
```
Require: $Q$: queue of incoming jobs; $Inputload$: number of clients for transactional
    or data size for MapReduce job; $P\_CLUSTER$: cluster of physical nodes;
    $V\_CLUSTER$: cluster of virtual nodes; $JCT_{desired}$: vector of jobs desired completion
    times.
 1: for each job $J_i$ in $Q$=$J_1$,$J_2$,...,$J_n$ do
 2:     if $J_i \in$ transactional workload then
 3:         Place $J_i$ on V_CLUSTER
 4:     else if $J_i \in$ batch MapReduce workload then
 5:         Profile $J_i$ using Algorithm 7 to obtain the vector of estimated job completion
            time ($JCT_{estimated}[]$).
 6:         if $JCT_{estimated}[i] \geq JCT_{desired}[i]$ then
 7:             Place $J_i$ on $P\_CLUSTER$
 8:         else
 9:             Place $J_i$ on V_CLUSTER
10:         end if
11:     end if
12:     return jobs assigned to $P\_CLUSTER$ and $V\_CLUSTER$
13: end for
```



(a) Job Profiling Error



(b) CPU Interference



(c) I/O Interference

Figure 5.8. (a) Profiling error in Phase I. (b), (c) show slowdown of JCT due to CPU and I/O interference from collocated VMs. Y-axis in (b) and (c) are normalized corresponding to the case when the job (`Sort`, `PiEst`) is run in isolation.

5.3.2 Phase II Scheduler

The goal of Phase II Scheduler is to efficiently manage the resources of the virtual cluster across transactional and batch MapReduce jobs, with an objective to comply with the SLAs of interactive jobs, while providing the best effort performance delivery to the MapReduce applications. Figure 5.9 shows the overall architecture of the Phase II Scheduler. It is composed of two main components: (i) *a Dynamic Resource Manager (DRM)* and (ii) *an Interference Prevention System (IPS).* The DRM monitors the available capacity on each VM to guide placement of MapReduce jobs within the virtual cluster. DRM records the resource consumption and completion times of each task run in the virtual cluster. This information is then used to build a model for each MapReduce job that correlates the resource allocation to the task completion time, allowing the scheduler to make intelligent decisions about where to place the remaining tasks of the job. The IPS is an online monitor that observes the performance of the interactive applications within the cluster to detect when they are not receiving sufficient resources to meet their demands. If this is detected, then the MapReduce jobs causing the interference are adjusted to reduce their impact. The VM running the task can have its resource share decreased, the VM can simply be paused, or it could even be migrated to a different host as part of different means to mitigate the observed interference. Even if the VM is fully stopped, the correctness of the corresponding MapReduce job is not affected, since the MapReduce master would regard this task as a prospective straggler and initiate its speculative execution [14].

To illustrate the level of interference which may be caused due to collocated VMs and/or contending tasks within the same VM, we conduct a study, where 4 VMs are deployed on a quad-core physical server, and run a mix of CPU and I/O bound MapReduce jobs. Each VM is pinned to a core, and 8 threads are run concurrently, sharing a single disk. We benchmark the completion time of CPU-bound `PiEst` and I/O-bound `Sort`. We first run each job on a single VM in isolation and note the JCT. We then measure the corresponding JCT when these jobs are run together with 3 other instances of `PiEst` and `Sort`, running on other VMs. Figure 5.8(b) shows the JCT of `PiEst` normalized to the corresponding JCT when run in isolation. The X-axis denotes the total CPU utilization, represented as the percentage of a single thread. We observe the slowdown in the run-time when the CPU interference from other background jobs increases, while I/O-bound `Sort` remains unaffected. Similar observation follows for I/O-bound `Sort` (Figure 5.8(c)), where we can see exponential increase in JCT due to

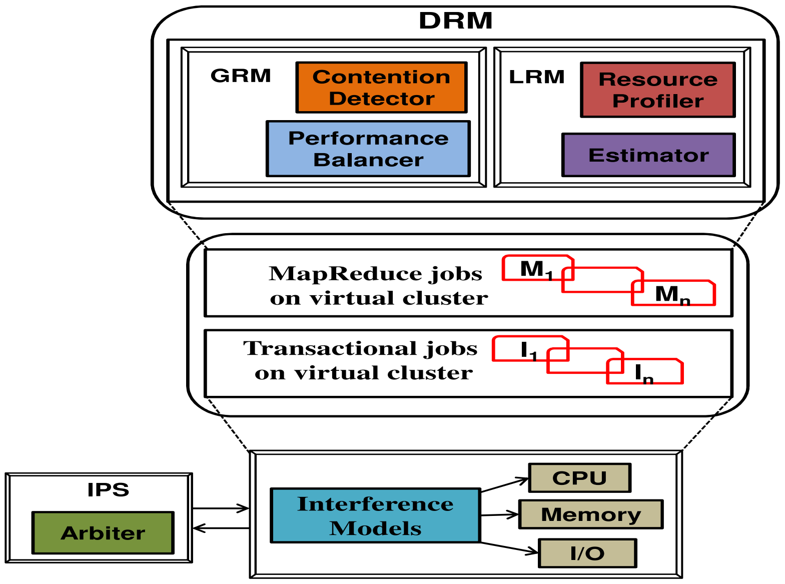


Figure 5.9. Components of the Phase II Scheduler.

increased I/O contention from other collocated VMs. Thus, understanding the run-time interference, and performing dynamic resource adjustment is required to mitigate such performance degradation. This approach is adopted by our Phase II Scheduler, whose details are described next.

5.3.2.1 Dynamic Resource Manager (DRM)

The goal of DRM is to build interference prediction models based on the run-time resource profiles of collocated tasks of MapReduce jobs and other co-hosted interactive jobs in order to estimate the task slowdown caused by resource contention. The estimations are used to orchestrate the optimal resource allocations via dynamic resource management to minimize the interference, and improve performance. DRM consists of two main components (a) a *Global Resource Manager (GRM)*; and (b) a *Local Resource Manager (LRM)*. The GRM consists of two sub-components: (i) a *Contention Detector* that dissects the cause of resource contention and identifies both resource-deficit and resource-hogging tasks and (ii) a *Performance Balancer* that leverages the run-time resource estimations from each LRM to suggest the resource adjustments based on the global coordinated view of all tasks. The LRM also consists of two sub-components: (i) a *Resource Profiler* that collects and profiles the run-time resource usage of tasks at each node and (ii) an *Estimator* that constructs statistical estimation models of a task's run-time performance as a function of its resource allocations.

DRM performs the following two functions: (i) Resource bottleneck detection – Here, the GRM based on the resource usage feedback from all LRMs identifies *which task* is experiencing bottleneck for *which resource*, on *which TaskTracker*. The rationale and scheme behind this detection mechanism is outlined in [124]. (ii) Resource bottleneck mitigation: The LRMs after getting information about both resource deficit and resource hogging tasks from the GRM, invoke their Estimator module to estimate the tasks completion times. The Estimator builds predictive models for the completion time of a task as a function of its resource usage/allocation. The difference between the estimated and the current resource allocation is the resource imbalance that has to be dynamically allocated to the resource deficit task.

The Estimator module is responsible for building predictive performance interference models for tasks' run-time resource usage/allocations (we focus on CPU, memory and I/O resources). In a virtualized environment, the interference is contributed by both co-located tasks within the same VM, and the co-hosted VMs on the same host. It employs statistical regression models to obtain the estimated optimal resource allocation for the next epoch of a task's run, which operates in two stages. The first stage inputs a time-series of progress scores of a given map/reduce task and outputs a time-series of estimated completion times corresponding to multiple epochs in its life time. The second stage uses a time-series of past resource usage/allocations across the task's run-time starting from its start till the current time of estimation. The estimation model, thus, outputs the estimated resource allocation for the next epoch as a function of its cumulative run-time. Separate estimation models are built for CPU, memory and I/O resource profiles of a task. We use the linear regression model for CPU, and piece-wise linear regression model for memory, as derived from our previous work [124]. In this research, we also estimate the I/O interference through an exponential regression model, similar to [87]. For controlling the I/O bandwidth allocation to individual task, we use the recently released Linux kernel support for I/O throttling using cgroups [125]. Complete details about the construction of models, along with the specifics of dynamic resource allocation mechanism, are detailed in our previous work [124].

Algorithm 9 Arbitration Algorithm.

```
1: Obtain the list of map/reduce tasks, $TASK\_LIST_{interference}$, interfering with
   collocated interactive jobs from the *Estimator*
2: Evaluate the estimated interference for each task in $TASK\_LIST_{interference}$ using
   prediction model (see Section 5.3.2.1)
   TaskInterference[]=GetEstimatedInterference() {Get list of all available VMs in
   $V\_CLUSTER$}
3: AvailableVMs[] = GetAllAvailableVMs()
4: **while** InterferenceTasks[] $\neq$ NULL **do**
5:    Get best available VM using BestFit bin-packing heuristic [78]
6:    $VM_{best}$=GetVMByBestFit()
7:    Use Min-Min heuristics [87] to schedule the 'least-interference' task on $VM_{best}$
8:    Adjust resources of $VM_{best}$
9: **end while**
```

5.3.2.2 Interference Prevention System (IPS)

For the transactional workloads, collocated with the MapReduce jobs in the virtualized environment, we build separate performance interference models, because their application characteristics are different from MapReduce jobs.We explored building both linear regression model as well as non-linear exponential regression model to quantify the CPU interference. However, we stick to a linear regression model as it fits well in our case in terms of offered simplicity and accuracy (see [124]). For memory, a piece-wise linear regression model well captures the interference [124]. For I/O, we use non-linear exponential regression based interference model. Specific details of the regression models used are similar to [85].

The IPS uses these performance interference models for interactive applications to dynamically orchestrate their resource allocations in the virtual cluster. The IPS continuously tracks the performance of interactive jobs. If at some instant, the performance falls outside allowable SLA bounds, the IPS invokes its *Arbiter* module to determine the specific task(s) and resource(s) that are causing this performance degradation. The Arbiter employs its arbitration scheme (Algorithm 9) to mitigate the interference, and restores the performance of the interactive applications. The Arbiter takes into consideration the interference from other collocated VMs, co-scheduled MapReduce jobs and VMs packing criteria, while deciding on the exact migration of VMs, or reassignment of interfering tasks to other VMs.

5.4 Experimental Evaluation

Infrastructure: Our experimental testbed contains a mix of physical and virtual nodes constituting native and virtual clusters, respectively. The *native cluster* contains 24 physical nodes. Each node has a dual 64-bit, 2.4 GHz AMD Opteron processor, 4 GB RAM, and Ultra320 SCSI disk drives, connected with 1 Gbps Ethernet. This cluster is installed with Hadoop v0.22.0 with 2 replicas per HDFS block, and 2 map, 2 reduce slots per node. The Hadoop FairScheduler [126] scheduling policy is used. The *virtualized cluster* contains 24 physical nodes which are virtualized with Xen v3.4.2 hypervisor [116], and hosts 2 VMs per PM to create an equivalent 48 nodes virtual cluster. Each VM runs RedHat AS4 with kernel v2.6.18, and the VMs are connected by 1 Gbps Ethernet. Each VM is configured with 1 GB RAM and 1 virtual CPU.

Benchmarks: We use a workload mix consisting of both transactional applications as well as batch workloads. For transactional workloads, we use three representative applications: (i) RUBiS [127], which models an online auction site; (ii) TPC-W [128], which models a three-tier online book store; and (iii) Olio [129], which is a Web 2.0 social events application. For batch workloads, we use the following representative MapReduce jobs.

- `Sort`*:* sorts 20 GB of text data (I/O-bound).
- `Wcount`*:* computes frequencies of words in 20 GB of text data (Memory + I/O-bound).
- `PiEst`*:* estimates Pi (10 million points) (CPU-bound).
- `DistGrep`*:* finds match of randomly chosen regular expressions on 20 GB of

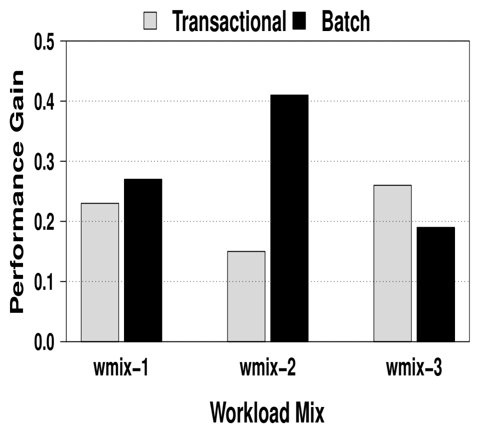
text data (I/O).

- `Twitter`*:* uses 25 GB twitter data [119] to rank users (Memory + I/O-bound).
- `Kmeans`*:* clusters 10 GB data (CPU-bound).

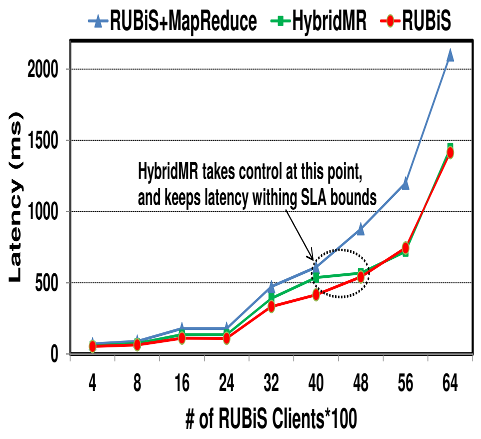
These MapReduce jobs are chosen based on their popularity and being representative of real MapReduce benchmarks, with diverse resource mix. We use the Yokogawa's WT210 power meter to measure server power. For better precision, we run all our experiments 3 times, and use the average value. Unless otherwise specified, all experiments are performed in the hybrid cluster consisting of 48 VMs over 24 PMs.

Performance Benefits of *HybridMR*: We begin with demonstrating the performance benefits of *HybridMR*'s Phase I Scheduler in the initial placement of MapReduce jobs. Towards this, we perform an experiment, where we measure the average response times of interactive applications and JCTs of MapReduce jobs, when scheduled by Phase I Scheduler, normalized against the corresponding JCTs measured with random placement of jobs (i.e., first-come-first-served discipline). In Figure 5.10(a), we plot this normalized JCT, labeled as 'Performance Gain'. We observe that through efficient placement by the Phase I Scheduler, both interactive and batch jobs benefit in performance, and the magnitude of gain varies depending on the workload mix characteristics. We consider 3 workload mixes: wmix-1, wmix-2 and wmix-3, where each represents 50% interactive + 50% batch jobs, 20% interactive + 80% batch jobs, and 80% interactive + 20% batch jobs, respectively. Intuitively, this performance gain is achievable because certain MapReduce jobs which are resource intensive and have more stringent performance bounds, when placed on native Hadoop cluster, reduces the interference across other MapReduce and interactive jobs.

We next analyze the potential benefits of Phase II Scheduler towards improving the performance of MapReduce jobs scheduled on the virtual cluster. In this experiment, we use 48 VMs virtual Hadoop cluster. We compare the performance of each of the six MapReduce jobs when run standalone (*Single*). We measure the JCT when Phase II Scheduler-based resource orchestration is enabled ($JCT_{hybridmr}$) versus without it ($JCT_{default}$). We compute percentage

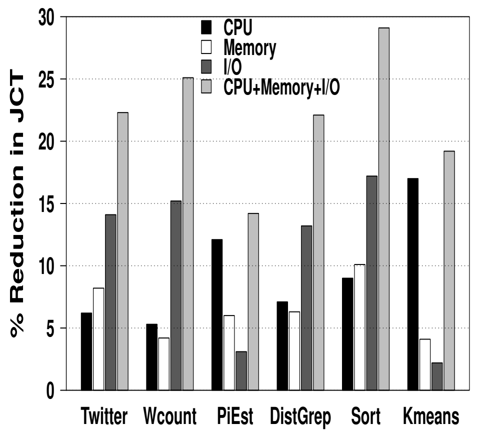


(a) Phase I scheduler placement

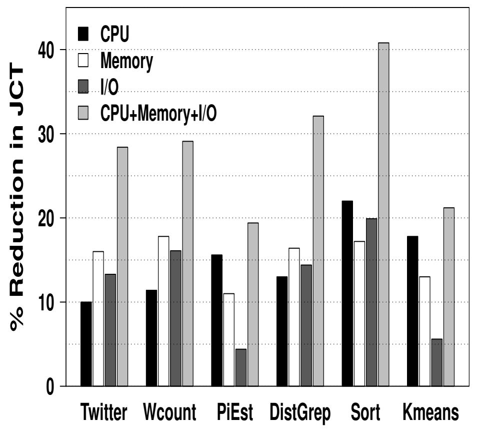


(b) SLA compliance with RUBiS

Figure 5.10. Performance benefits of *HybridMR*.



(a) Single job improvement



(b) Multiple jobs improvement

Figure 5.11. Performance benefits of *HybridMR* on a virtualized platform.

reduction in JCT as $(JCT_{default} - JCT_{hybridmr})/JCT_{default}$*100. Figure 5.11(a) shows the percentage reduction in JCT of the six MapReduce jobs. Here, each legend corresponds to the case when only CPU, Memory, I/O or all three of them are managed by *HybridMR*. Across all jobs, *CPU+Memory+I/O* yields the maximum reduction in JCTs, while the reduction with *CPU*, *Memory* or *I/O* alone varies in accordance with the benchmark resource characteristics. Specifically, we observe an average and a maximum JCT reduction of 22% and 29.1%, respectively, with *CPU+Memory+I/O* mode. Note that larger jobs (with respect to both large input size and long running job) like `Sort`, `Kmeans` benefit more when compared to other relatively shorter jobs like `PiEst`, `DistGrep`. This is due to the fact that larger jobs run in multiple map/reduce phases, and thus, *HybridMR* has more opportunities to dynamically coordinate resources and improve their JCTs. Similar benefits are observed in the JCT of each job when run in the presence of other 5 concurrent running MapReduce jobs (*Multiple jobs*). Figure 5.11(b) shows the results. Note that since *Multiple jobs* scenario induces more inteference than corresponding *Single job*, more performance benefits are observed with *Multiple jobs*. We observe an average and a maximum reduction of 28.5% and 40.8% in the JCTs with *CPU+Memory+I/O*. In Figure 5.12, we also observe high CPU, memory and I/O utilization achieved by *HybridMR*.

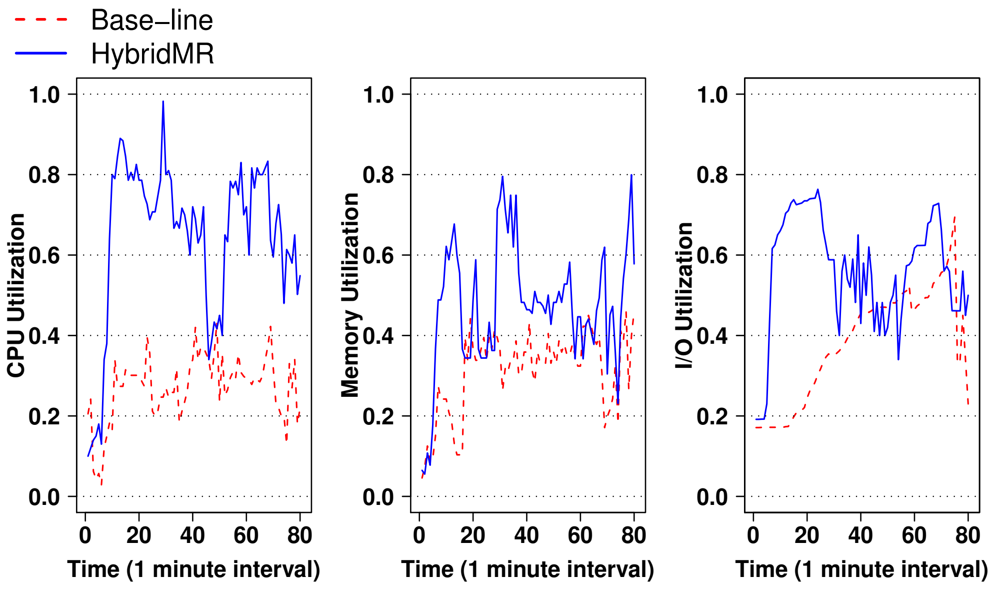
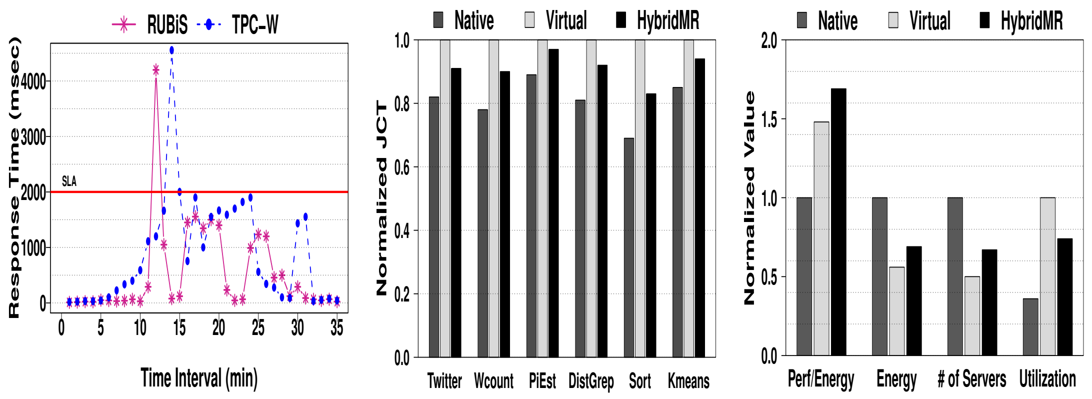


Figure 5.12. *HybridMR* improves the resource utilization of MapReduce clusters.

Next, we analyze the impact of *HybridMR* on the SLAs of interactive applications like RUBiS. We perform an experiment, where we increase the number of RUBiS clients and observe the impact on the latency of web server. Figure 5.10(b) shows the results. Here, the top *RUBiS+MapReduce* curve corresponds to the case where the RUBiS virtualized cluster co-hosts other MapReduce jobs assigned in the FIFO order by the default MapReduce scheduler. The bottom *RUBiS* curve denotes the case where RUBiS runs in isolation with no MapReduce jobs. The middle *HybridMR* curve depicts the scenario with *HybridMR* scheduler. Here, we observe that for low client



(a) Performance gain (b) JCT improvement (c) Potential savings

Figure 5.13. *HybridMR* achieves better balance between Native and Virtual. Y-axis in (b) and (c) is normalized with respect to the maximum.

workloads, *HybridMR* is able to co-host MapReduce jobs with RUBiS and keep its latency within bounds. However, when *HybridMR* detects that client latency is exceeding a given threshold, its Phase II Scheduler can adaptively migrate or pause the execution of co-hosted interfering MapReduce tasks to bring down the latency similar to the bottom *RUBiS* curve. This shows the dynamic adaptation of *HybridMR* in keeping up with the SLA of interactive applications.

In a related analysis, this chapter demonstrates the effectiveness of *HybridMR* in maintaining the SLAs of interactive RUBiS and TPC-W applications, when they run collocated with other MapReduce jobs. In Figure 5.13(a), we observe that in the time interval 1-10 minutes, the response times of both RUBiS and TPC-W are below the user defined SLA (2 seconds in our experiment). However, around 12^{th} (RUBiS) and 14^{th} (TPC-W) time instant, the response time exceeds SLA. *HybridMR* quickly identifies this, and is able to migrate the collocated map/reduce tasks from the VMs running RUBiS and TPC-W to mitigate the interference. Consequently, the response time falls within limits again. Similar behavior follows for Olio application.

Cross-Platform Performance Comparison: We next demonstrate the benefits of *HybridMR* in scheduling a mix of interactive and batch jobs. In this experiment, we evaluate three design choices for cluster configuration. In the first design choice (*Native*), the workload mix is scheduled on a 24-node native

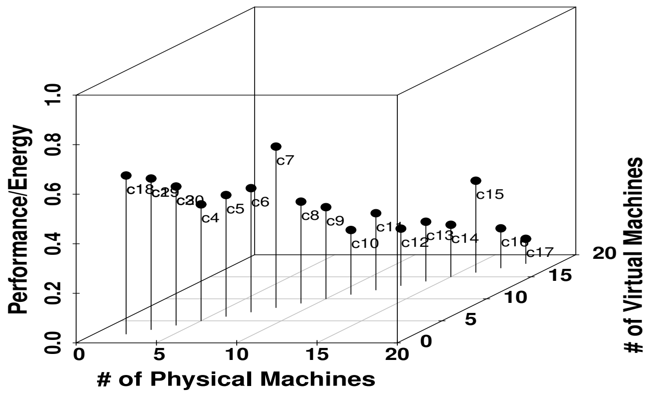
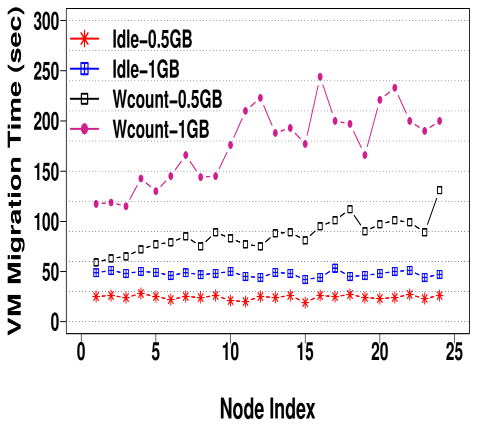


Figure 5.14. Hybrid configuration design trade-off analysis.

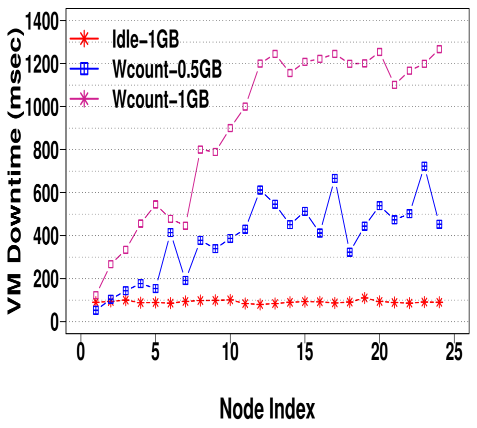
Hadoop cluster. In the second design choice (*Virtual*), the workload mix is scheduled on a 24-node virtual cluster (hosted on 12 PMs, each PM contain 2 VMs). The third design choice (*HybridMR*) corresponds to the case, where the workload mix is scheduled on a 24-node hybrid cluster, consisting of 12 PMs + 12 VMs (consolidated on 6 PMs with 2 VMs each). Thus, *Native*, *Virtual* and *HybridMR* requires 24, 12, and 18 different number of physical servers, respectively. From Figure 5.13(b), we observe that the performance of MapReduce jobs is the worse for *Virtual* because of the possible virtualization overheads. *Native* achieves the best performance as expected. *HbyridMR*'s performance is intermediate between *Native* and *Virtual*. However, as we observed in Figure 5.13(c), *HybridMR* achieves the highest *Performance/Energy* which is an important design metric. In terms of other metrics like energy, utilization, and number of physical servers required, *HybridMR* achieves a better balance between the two extreme choices – *Native* and *Virtual*.

Design Trade-off Analysis: We next discuss the flexibility of *HybridMR* in terms of splitting a given physical infrastructure into different hybrid cluster configurations, each consisting of a fixed number of VMs and PMs for running a workload mix for different performance and energy requirements. To highlight this empirically, we split our testbed (consisting of 24 PMs hosting 48 VMs) into 20 different cluster configurations ($C_1 - C_{20}$), where each C_i contains a randomly selected number of PMs and VMs, and runs workload mix of interactive and MapReduce jobs. For each configuration, we measure the average completion times (performance), average energy consumed and average utilization across all jobs and nodes. In Figure 5.14, each configuration corresponds to a particular $\langle \#ofPMs, \#ofVMs, Performance/Energy \rangle$ tuple in the 3D space. For example, configuration C_7 (with 12 PMs, 12 VMs) and C_{17} (with 24 PMs, no VMs) gives the best and worst *Performance/Energy*, respectively. Such analysis can provide some useful insights. For example, a cluster administrator may use some hints from a similar empirical analysis, while deciding how to partition the infrastructure into hybrid cluster to meet a desired performance-energy envelope.

Impact of live migration of Hadoop VMs: In the IPS module (Section 5.3.2.2), we leverage live migration of VMs as a mechanism to migrate interfering map/reduce tasks. To understand the performance implications of Hadoop VMs migration, we perform some empirical analyses. We run `Wcount` on different data size on a 24 VMs Hadoop cluster, and randomly migrate the 24 VMs during the job execution. Figure 5.15(a) and Figure 5.15(b) show the migration time and downtime of each VM during migration. We observe: (i) larger the amount of memory involved, longer is the migration time, while the downtime dependence on memory is ad-hoc; (ii) migration time of VMs running `Wcount` is relatively longer than the corresponding idle Hadoop cluster; and (iii) downtime of each VM running `Wcount` shows wide variation, possibly due to skewness of each Hadoop VM, due to difference in resident data blocks, concurrent map/reduce tasks, and interference from other collocated VMs. From this analysis, we conclude that live migration of Hadoop VMs incurs some overheads, especially the downtime. However, due to the inherent fault-tolerant characteristics of Hadoop (*i.e.,* replication), the downtime period can be balanced by regenerating the data blocks from other available copies. However, the MapReduce jobs still run to finish in the face of the migration induced downtime.



(a) VM migration



(b) VM downtime

Figure 5.15. Illustration of the performance impact of live migration of Hadoop VMs.

5.5 Chapter Summary

This chapter presents a 2-phase hierarchical scheduler, called *HybridMR*, for hybrid data centers, consisting of a mix of native and virtual machines to leverage the benefits of both paradigms. In the first phase, *HybridMR* profiles incoming MapReduce jobs to gauge the estimated virtualization overheads, and utilizes this information to automatically guide placement between physical machines and virtual machines. In the second phase, *HybridMR* builds run-time resource prediction models, and performs dynamic resource orchestration to minimize the interference within and across collocated interactive and MapReduce applications. Detailed evaluations on a hybrid compute cluster consisting of 24 physical servers and 48 virtual machines, with diverse workload mix of interactive and batch MapReduce benchmarks, demonstrate that *HybirdMR* achieves up to 40% improvement in job completion times of MapReduce jobs over a virtual cluster, while satisfying the SLAs of interactive applications. Furthermore, *HybridMR* provides 45% improvement in resource utilization and around 43% energy savings compared to a native cluster, with minimal performance penalty. Moreover, we empirically demonstrate that it is possible to dynamically change the native and virtual cluster configurations to accommodate variations in workload mix for maximizing the performance-energy envelope. These results establish that a hybrid cluster configuration is a better cost-effective solution than the either-or approach to native and virtual modes of computation.

Chapter 6

Problem Determination and Diagnosis in Virtualized Clouds

6.1 Introduction

Large data centers and utility clouds experience frequent faults, which are top contributors to their management costs, and lead to Service Level Agreement (SLA) violations of the hosted services [130, 131, 132, 98]. For example, recently, Amazon EC2 experienced a power outage that brought down their servers for seven hours, incurring severe financial penalties [133]. The existence of public repository of failure traces [19] across diverse distributed systems like Skype, Microsoft, Planetlab, clearly demonstrates the prevalence and need for taming the faults for successful operation. A recent survey [21] demonstrates increasing customer reluctance to move to clouds due to poor performance. Another study [20] on 3 years worth forum messages concerning the problems faced by end users of utility clouds shows that virtualization related issues contribute to around 20% of the total problems experienced. These faults contribute to the poor performance and SLA violations of the applications. [106, 99, 134, 135].

Traditional problem determination in distributed systems is geared towards building a model of an application running without errors [130]. When an

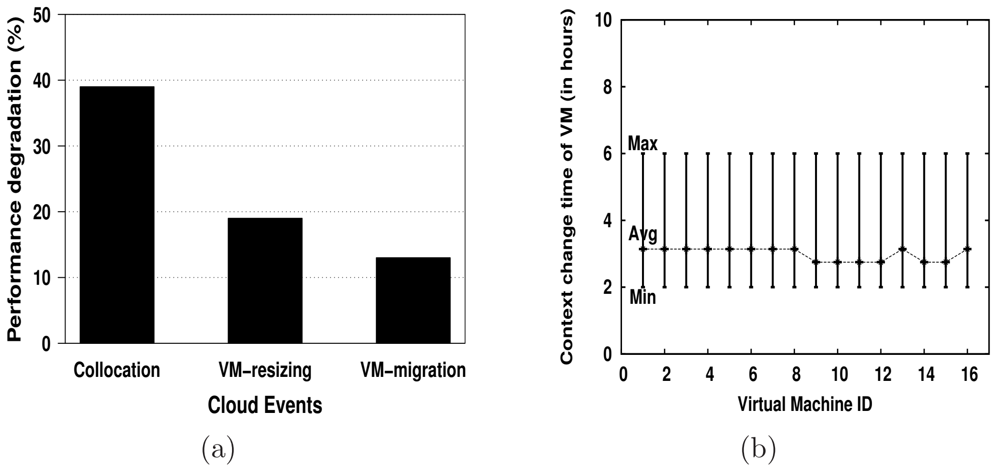


Figure 6.1. (a) We ran a file system benchmark, *Iozone*, and observed degradation in its completion time due to various cloud events. (b) Operating context of workloads changes frequently due to high dynamism in clouds.

application's current performance does not match the model of its normal execution state, an anomaly is detected, thereafter system administrators are alerted, who typically fix the anomaly manually. Clouds present an automated and dynamic model, which conflicts with the manual/semi-automatic process of problem determination. In clouds, there is a growing trend of increasing number of performance anomalies [19, 20]. The applications running inside the cloud often appear opaque to the cloud provider, which makes it non-trivial to get access to fine-grained system and application measurements for problem detection and diagnosis [106]. Thus, in particular, problem determination and diagnosis in clouds is very challenging and introduces new issues, which we explain them below.

6.1.1 Problem Determination in Clouds: What is New?

In this work, we address the issue of problem determination in a multi-tenant dynamic Infrastructure as a Service (IaaS) cloud environment. We focus only on problems that lead to performance degradation, but do not cause the application to fail to execute altogether. In addition to large scale, clouds present the following new challenges that traditional problem determination techniques fail to meet.

- *Sharing of Resources*: Clouds are multi-tenant, and multiple virtual machines (VMs) are collocated on the same physical server. Since resources like cache, disk and network bandwidth are not virtualized, the performance of an application may depend on other co-hosted applications. We conducted a preliminary study to understand the impact of collocation (multiple applications sharing the same set of resources), VM migration and VM resizing. We observed that multi-tenancy can lead up to a 40% performance degradation for a sample file system benchmark *Iozone* (Figure 6.1(a)). This makes it important for problem determination techniques to understand the *operating context* under which a workload operates and distinguish a collocation fault from an application error. Operating context quantifies the impact of collocated applications by augmenting the metrics that are affected by the environment (*e.g.,* host metrics like cache miss, page fault) in the application performance model. We elaborate on the notion of operating context in Section 6.3.
- *Dynamism*: Clouds are elastic and allow workloads to automatically request/release resources. Elasticity in a cloud is enabled using VM resizing, VM migration, and VM cloning. Hence, the operating context under which a workload operates, changes more frequently as compared to traditional distributed systems. In a case study (see Section 6.5.6), we observed that the operating context of all VMs changes within 3.5 hours (Figure 6.1(b)), and the maximum duration without a change in operating context for any VM was 6 hours. This makes it imperative for a problem determination system to dynamically learn the application behavior in the specific operating context, rendering static model-based approaches ineffective for clouds [96, 136, 99].
- *High Frequency of Faults*: Sharing of resources combined with high dynamism invariably leads to a large number of cloud anomalies [105, 106, 99, 20]. We observed that a faulty VM resizing can impact performance by up to 20% and a faulty VM migration can impact performance by more than 10% (Figure 6.1(a)). Further, we found that up to 10% of cloud reconfiguration actions can be faulty (see Section 6.5.6). Hence, problem determination in clouds needs to deal with a much higher frequency of faults than traditional

distributed systems.

- *Autonomic Systems*: Cloud is an autonomic end-to-end system, which reacts automatically to changes in workload requirements [106]. A static problem determination system that flags a VM sizing error and waits for manual intervention does not fit the cloud model [96, 106, 105]. Problem determination in clouds needs to take a complete automated end-to-end approach for anomalies detection, diagnosis, classification and remediation by integrating with the cloud management stack [106]. To the best of our knowledge, no such framework exists today for an automated end-to-end handling of cloud related faults. Thus, we present the design and implementation of a comprehensive end-to-end fault management framework, called *CloudPD*, for virtualized cloud platforms.

The design of *CloudPD* includes the operating context of a workload in its resource model to quantify the performance impact of collocated applications in a scalable manner. *CloudPD* consists of online model generation techniques for building performance models for applications by considering them simply as a black box. It combines simple and correlation-based models in a two-phase methodology, where a light-weight resource model is first used to predictably trigger events, followed by a correlation-based analysis only for a sub-set of these events. To summarize, we make the following contributions in this work:

6.1.2 Contributions

- We show that high dynamism and sharing of resources in the cloud often lead to frequent changes in an application's resource model, obviating the applicability of problem determination techniques that create a stationary model for a system and identify deviations from the model to flag errors. Further, cloud introduces new anomalies due to sharing and cloud reconfiguration, which are difficult to differentiate from application errors. We identify the need for an end-to-end problem detection, diagnosis and remediation framework, consistent with the autonomic cloud management system. Specifically, we focus on anomalies that arise due to cloud activities and virtualization artifacts such as VM migration, VM resizing, and VM collocation, unlike prior work, where the thrust was primarily on application level anomalies.

- We present the design and implementation of *CloudPD*, a fault management framework, which addresses the challenges identified with a problem determination framework for clouds. *CloudPD* introduces three novel ideas and combines them with known techniques to design an effective methodology for problem determination. Our first idea attacks the problem of a non-stationary context by introducing *operating context* of an application in its resource model. The key idea is in using host metrics as a canonical representation of the operating context for drastically reducing the number of resource models to be learned. Moreover, we use an online learning approach to further reduce the number of resource models learned by the system. The third idea is a three-level framework (*i.e.,* a light-weight event generation stage, an inexpensive problem determination stage and a robust diagnosis stage) which combines resource models with correlation models as an invariant of application behavior. Since pair-wise correlation behaviors are expensive to learn, we restrict ourselves to learning (i) only linear correlations and (ii) correlations between metrics that exhibit affinity, allowing the system to scale efficiently.
- We have implemented a working prototype of *CloudPD*, and performed comprehensive evaluations on 28 VMs with cloud representative benchmarks – Hadoop, Olio, and Rubis, where *CloudPD* diagnosed faults with low false positives and high accuracy of 88%, 83%, 83%, respectively. In another real enterprise trace-driven case-study on an IaaS cloud testbed, *CloudPD* with fewer false alarms, achieved high recall, precision, and an accuracy of 77%. Furthermore, *CloudPD* can suggest the required remediation actions to the cloud resource manager within 30 seconds.

The rest of this chapter is organized as follows. Section 6.2 outlines certain key design goals for our system. The *CloudPD* architecture is detailed in Section 6.3 and implementation details are discussed in Section 6.4. In Section 6.5, we describe our experimental platform and present results from extensive evaluations. The summary of this chapter is briefed in Section 6.6.

6.2 Background

In this section, we discuss how *CloudPD* interfaces with a cloud ecosystem, and addresses the new challenges posed by virtualized cloud environments.

6.2.1 End-to-end Problem Determination in a Cloud Ecosystem

IaaS cloud management systems are autonomic and continually optimize the cloud infrastructure to adapt to workload variations. Since future management actions in a cloud are dependent on the outcome of previous actions, the cloud management stack should be made aware of any faults that happen as a result of these events. Hence, a problem determination system in a cloud needs to be autonomic and provide fault remediation actions for cloud related faults.

Figure 6.2 captures the system context in which a cloud problem determination system needs to operate. A Cloud Resource Manager (CloudRM) periodically reconfigures the cloud using the monitored system, application data, and workload profiles. The monitored data is used to estimate the resource requirements and the workload profiles are used to identify workloads that may conflict with each other. The reconfiguration events are passed to the virtualization layer for further actions.

There are two aspects of a cloud ecosystem that merit attention. First, IaaS clouds use multi-tenancy to ensure that compute resources are utilized optimally. Multi-tenancy on a virtualized server is supported by reserving CPU and memory resources for individual VMs. However, virtualization does not allow reservation of resources that are not traditionally managed by operating systems, such as cache, memory, network and I/O bandwidth. Due to the shared nature of these resources, an application may witness a change in performance if VMs are either added or removed from a physical server hosting the workload. Hence, clouds introduce

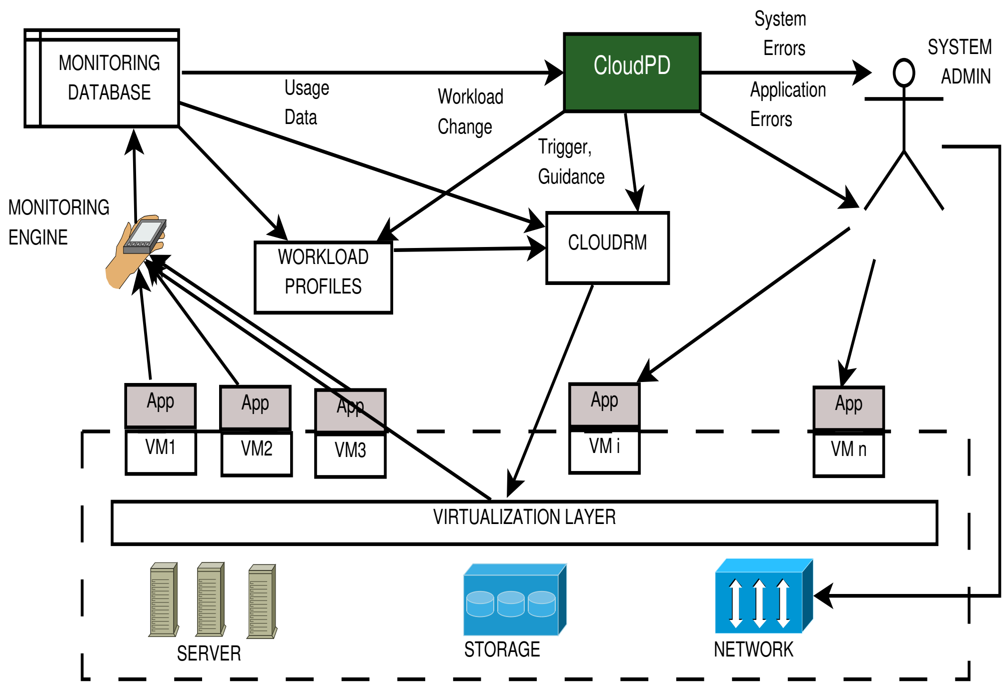


Figure 6.2. System Context for *CloudPD* System.

collocation faults, where an application experiences a performance anomaly due to collocated VMs.

The second aspect that is of interest is the continual reconfiguration in a cloud. Two particular reconfiguration actions are relevant in terms of performance anomalies. CloudRM uses prediction to estimate the size of the VMs that host a workload [137] and resizes a VM based on the estimate. An error in prediction may lead to a VM being allocated fewer resources than it requires, leading to a VM resizing fault. Similarly, CloudRM uses VM live migration to ensure that all workloads are hosted on as few servers as possible at any time instant. VM live migration can similarly lead to performance degradation [138]. During VM live migration, all memory pages are marked as ready-only and writes lead to faults. Hence, the performance of write-intensive workloads can suffer during live migration. Further, live migration requires significant amount of CPU and this can lead to performance impact on live migrations as well as failed live migrations.

Clouds introduce new fault types that need to be distinguished from traditional application faults. However, they pose more fundamental challenges for problem determination. Problem determination in traditional distributed systems has always focused on *one application at a time* [19]. Collocation faults imply that the model for normal behavior of an application needs to be learned in the context of other co-hosted applications. Combined with VM live migration, it implies that this model needs to be learned in the context of *all* possible sets of applications that can be collocated with our target application. In a cloud with thousands of applications, creating such a large ensemble of models is clearly infeasible. Similarly, since resources allocated to a VM can change (by dynamic VM resizing), none of the thresholding techniques [136] can be applied. Correlation based models are typically more stable and can be applied. However, the number of relevant pair-wise correlations are exponential in a cloud, where any application can be impacted by any other collocated application. Finally, a cloud problem determination system (*e.g., CloudPD*) needs to integrate with the cloud management infrastructure and update any collocation errors as changes in workload profile. If any VM has been wrongly sized, *CloudPD* needs to trigger CloudRM to estimate the new sizes for the VM. Similarly, any faulty live migrations should be communicated as guidance to CloudRM for future configuration actions. Application and system errors are handled by sending emails to system administrators. The cloud ecosystem imposes a need for not just fault detection but diagnosis, classification and automated remediation.

6.3 Architecture

6.3.1 Design Decisions

CloudPD is designed as an autonomic end-to-end problem detection, diagnosis and remediation framework, which synergizes with the cloud management system to guide remedial actions for dealing with various cloud-based faults. *CloudPD* is based on the following three key design decisions:

- **Include Operating Context in the Performance Model**: One of the key ideas in *CloudPD* is to include the *operating context* of a workload in its performance model to capture the impact of collocated applications in a scalable fashion. The most natural choice for operating context is the set of collocated applications for each VM. However, the number of possible operating contexts for just one application is exponential in the number of *other* VMs in the cloud (*e.g.,* 1000^5 for VM density of 6 with 1000 VMs). The hypothesis which we made here is that including only host metrics in the operating context may reasonably approximate the impact of collocated VMs. If multiple VMs have the same collocation impact on our target application, this idea merges all VMs into a common operating context, drastically reducing the number of operating contexts. We define an operating context for a VM to include the (i) host metrics and (ii) impacted metrics. Impacted metrics are defined as those which are affected by the environment and include L1/L2 cache misses, context switches, page faults, etc. Host metrics include the resource metrics (CPU, memory) and impacted metrics for the physical server hosting the VM. Figure 6.3 captures an example of the effectiveness of this canonical representation. For both an anomalous VM migration interval and a normal interval, the CPU and memory utilization of a VM are nearly the same. However, the distinction that allows *CloudPD* to identify a migration fault is primarily in the cache misses experienced at the server hosting the VM.

- **Online Model Generation**: Traditional problem determination learns an application model *a priori* based on historical data. Even with our transformation, the number of canonical operating contexts may be large and difficult to be known *a priori.* Hence, *CloudPD* follows an online model generation approach for building performance models. This also allows *CloudPD* to consider the applications as a black box and extend to new ones on the fly.
- **Combining Simple and Correlation-based Models**: Simple resource models are less expensive but suffer from poor stability in a dynamic environment. We observed that pair-wise correlation values are stable (Section 6.5.4) but learning correlations for all pairs is fairly expensive. We combine the strengths of both these techniques in a two-phase approach,

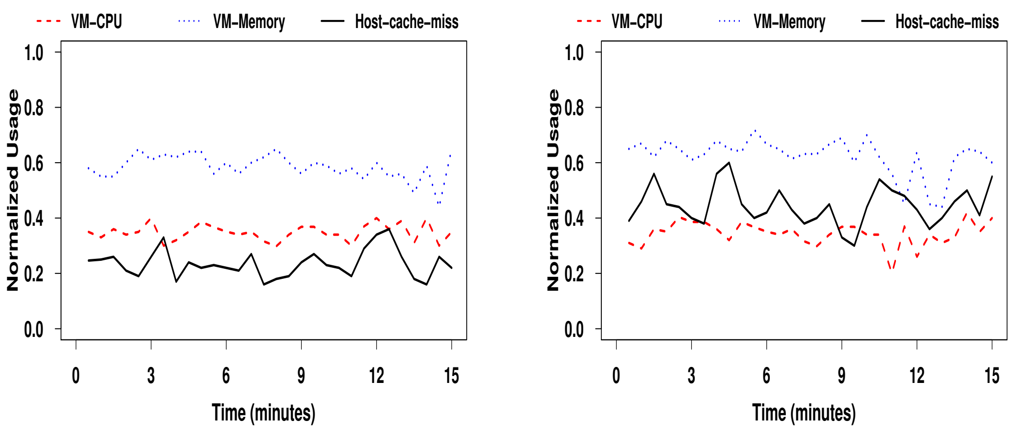


Figure 6.3. Importance of operating context (a) VM CPU, memory and host cache misses for a normal interval (b) Higher host cache misses, while VM CPU, memory remain normal for an anomalous VM migration event. We see a 95% difference in the average usage of host cache-miss across (a) and (b).

where a light-weight resource model is used to generate events in the first phase and moderately expensive correlation-based analysis to accurately identify faults is performed only for a small number of intervals, where the first phase generated an event. In order to further reduce the time spent in the second phase, we (i) use only linear correlations, which are present for 50% metrics [139], and can be learnt quickly with small amount of data and (ii) compute correlations only for those metric pairs, which are likely to be correlated (i.e., metrics of the same VM or same metric for VMs that are part of a cluster). Computing correlations for only a subset of intervals and a subset of pairs helps *CloudPD* scale to large data centers. We also considered the use of canonical correlation analysis (CCA) to quickly identify relevant correlated metrics between any pair [99] to further reduce the time. However, CCA can only extract positive correlations and we also observed negative correlations in practice (*e.g.,* CPU and disk activity were found to be negatively correlated) preventing its use in *CloudPD*.

6.3.2 Architecture

The architecture of *CloudPD* is described in Figure 6.4. It comprises of five key components: (a) a *Monitoring Engine* that monitors each virtual machine and physical server for resource metrics, application metrics and the operating context; (b) an *Event Generation Engine* that quickly identifies a potential symptom of a fault and generates an event; (c) a *Problem Determination Engine* that further analyzes the event to determine deviations from normal behavior; (d) a *Diagnosis Engine* that classifies the anomaly based on expert knowledge; and (e) an *Anomaly Remediation Engine* that executes remedial actions on the diagnosed anomalies. We next describe each of these in detail.

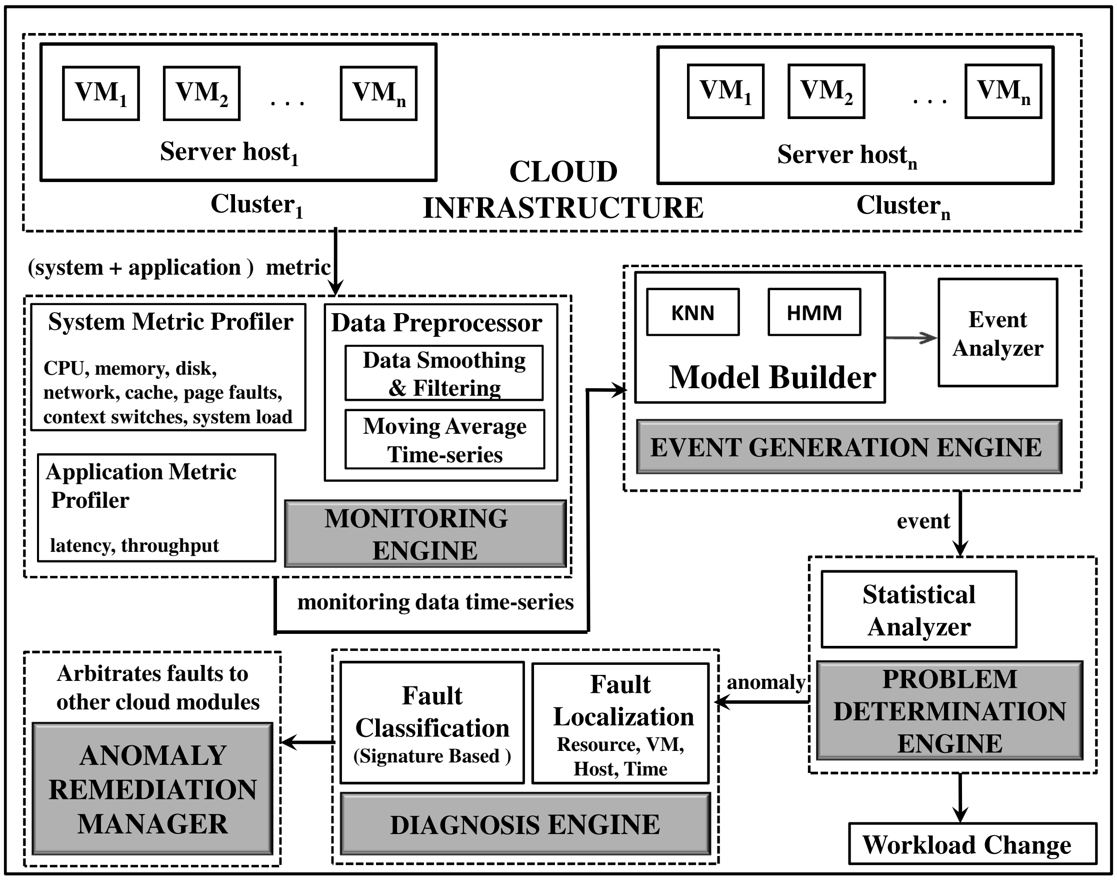

VM1
VM2
VMn
Server host1
Cluster1
CLOUD INFRASTRUCTURE
Server hostn
Clustern
(system + application) metric
System Metric Profiler
CPU, memory, disk, network, cache, page faults, context switches, system load
Data Preprocessor
Data Smoothing & Filtering
Moving Average Time-series
Application Metric Profiler
latency, throughput
MONITORING ENGINE
monitoring data time-series
KNN
HMM
Model Builder
Event Analyzer
EVENT GENERATION ENGINE
event
Statistical Analyzer
PROBLEM DETERMINATION ENGINE
Workload Change
anomaly
Fault Classification (Signature Based)
Fault Localization Resource, VM, Host, Time
DIAGNOSIS ENGINE
Arbitrates faults to other cloud modules
ANOMALY REMEDIATION MANAGER

Figure 6.4. Architecture of *CloudPD*.

6.3.2.1 Monitoring Engine

This component collects and processes various system metrics pertaining to CPU, memory, cache, network, and disk resources, and application metrics such as latency and throughput for every virtual machine and physical server. It is designed to capture (i) system resource metrics, (ii) operating context, and (iii) application performance metrics. A *System Metric Profiler* collects system resource metrics as well as the operating context, while the *Application Metric Profiler* collects all the application performance metrics. All metrics are collected at periodic intervals with a configurable monitoring interval parameter. *CloudPD* has a *Data Preprocessor* module that removes outliers and noise. The *Data Preprocessor* takes the raw time-series data and generates *data-points*. A *data-point* is defined as a sequence of moving average values over a fixed interval of time and forms the basic input unit for our anomaly detection algorithms.

6.3.2.2 Event Generation Engine

This module implements the first phase of our multi-layered problem determination engine. It is designed to identify potential symptoms of anomalous behavior without performing a lot of computation. The generation of an event may not immediately denote the presence of a fault, but merely suggests the possibility of one. It indicates that one or more metrics for a VM deviates from its performance model (the model for normal behavior is continuously generated online to cope with dynamic changes in the workload and operating environment). Further analysis would be needed to confirm if a fault is present, and if so, diagnose and classify it.

In order for *Event Generation* to be light-weight, events are generated by looking at each monitored metric for each VM in isolation. A broader correlation between metrics across VMs can be very expensive as the number of such comparisons grow exponentially in $N \times M$, where N is the number of VMs and M is the number of monitored metrics for each VM. Note that the *Event Generation* can be parallelized with a separate thread performing the analysis for each (metric, VM) pair. Further, in order to build the performance model in an online fashion that is tolerant to workload variations, we use the nearest-neighbor algorithm as it is (i) simpler to implement; (ii) computationally less expensive when compared to other techniques such as clustering; and (iii) can be updated online at very little cost [140]. We leverage existing techniques [141, 130, 106] for the methodology used in this phase (see Section 6.4.2).

6.3.2.3 Problem Determination Engine

This component uses statistical correlation across VMs and resources to identify anomalies and localize them. For every event generated by the *Event Generation Engine*, this stage analyzes the data further to identify anomalies (the *Problem Determination Engine* is not invoked unless an event is generated). Let us suppose that an event was generated for metric M_j on VM VM_i. This stage then computes correlations between data for M_j and data for every other metric on VM_i, as well as correlations between data for M_j on VM_i and data for M_j on every other VM running the same application (wherever the information is available). The first set of correlations capture the relation between metrics for the VM (*e.g.,* the correlation between resource metrics and their operating context), whereas the second set of correlations is based on the idea of using peer VMs to flag faults that occur in only one VM. Based on the knowledge of typical correlation values under normal behavior, any significant deviations from normal correlation values (with range [-1, 1]) are noted. If the deviations are larger than a threshold, a fault event is generated and forwarded to the *Problem Diagnosis Engine* for problem diagnosis; otherwise, when no anomaly is raised, that event is flagged as a normal workload activity (see Algorithm 10 for details). Please note that although our approach is similar to the idea of peer based correlation [99], we only perform correlations on events generated in the *Event Generation* stage, and only for the VM(s) and resource(s) tagged in this phase. This helps improve the scalability significantly.

In order to cope with dynamism, the *Problem Determination Engine* phase computes models and the deviations in correlation described above in an online manner as shown in Algorithm 10. Correlations on data from an interval classified as normal from recent history is used to obtain a model of normal application

Algorithm 10 Correlation-based Problem Determination.

```
1: Let $T_{normal-vm-r}$ = Data for VM $vm$ and metric $r$ over a normal interval from recent
   history; $T_{test-vm-r}$ = Data for VM $vm$ and metric $r$ combined over test interval and
   normal interval
2: for each VM $i$ in cluster do
3:     if $(ABS(corr(T_{test-vm-r}, T_{test-i-r}) - corr(T_{normal-vm-r}, T_{normal-i-r})) \geq$
       $Threshold)$ then
4:         Flag deviation as anomaly for diagnosis
5:     end if
6: end for
7: for each Metric $j$ do
8:     if $(ABS(corr(T_{test-vm-r}, T_{test-vm-j}) - corr(T_{normal-vm-r}, T_{normal-vm-j})) \geq$
       $Threshold)$ then
9:         Flag deviation as anomaly for diagnosis
10:    end if
11: end for
12: if No anomaly is flagged then
13:     Flag as normal workload change
14: end if
```

behavior. For the interval being analyzed, we compute similar correlations and check if these correlations deviate from the model of normal behavior. The total number of correlations computed in this algorithm is of the order of the number of metrics plus the number of VMs running the application. Note that we do not analyze the parts of the system that are not affected, and only analyze the *neighborhood* of the location where an event is generated. This allows *CloudPD* to scale in terms of the large number of metrics monitored.

6.3.2.4 Diagnosis Engine

This component uses pre-defined expert knowledge to classify the potentially anomalous scenarios detected by the *Problem Determination Engine* into one of several faulty and non-faulty classes. The expert knowledge is made available to the system in the form of standardized *fault signatures*. Characterization of system anomalies in terms of representative signatures is a critical part of diagnosis process [142]. The fault signature captures the set of deviations from normal behavior, both in the correlation values computed by the *Problem Determination Engine* as well as in the operating environment, that are characteristic of the fault they describe. When anomalous behavior is detected, the deviations from normal behavior are matched with the known fault signatures. If a match is found, the *Diagnosis Engine* will successfully classify the fault, else flag it as an anomaly.

6.3.2.5 Anomaly Remediation Manager

This component of *CloudPD* receives input from the *Diagnosis Engine* to perform suitable remedial actions. It is designed to deal with all the cloud related faults identified. In case of a collocation fault, *CloudPD* sends an exclusion rule to CloudRM that prevents collocation of the impacted VMs on a common physical server. In case of a faulty live migration, it provides new server utilization thresholds beyond which live migration should not be performed. In case of a VM sizing fault, it triggers resource estimation and resizing by CloudRM. For all other cases, a notification is sent to an application administrator or a system administrator. Note that, although it is true that automated remediation is often unacceptable to system administrators, especially for traditional distributed systems like grids and data centers, clouds are forcing a paradigm shift. Clouds represent a much more dynamic system, which employ a dynamic consolidation manager like VMware DRS or Amazon AutoScaling to *automatically* deal with unexpected performance changes by live migration or cloning VMs. In comparison, *CloudPD* actions are safer (triggers cloud manager to re-consolidate or add exclusion constraints between conflicting VMs only).

6.4 Implementation Details

We have implemented *CloudPD* to perform fault diagnosis for a VMware ESX-based cloud cluster (details are in Section 6.5). We provide specific details of our implementation below.

6.4.1 Monitoring Engine

CloudPD's *Monitoring Engine* collects measurements from individual VMs as well as physical servers, for fault detection and diagnosis. The measurement data (reported in Table 6.1) include basic resource metrics (CPU, memory), impacted operating context parameters (context switches, cache misses), host operating context parameters, and application performance metrics (latency, throughput). We use Linux *sar* utility to collect the VM level system metrics and VMware $vmkperf$ [143] performance monitoring tool for obtaining server's cache misses. To collect CPU and memory usage of the server, we use VMware *powercli cmdlets* [144]. The monitored data across all VMs and servers are collected via remote network copy, and stored at a central VM for further processing by subsequent stages. The *Data Preprocessor* module generates data-points from the raw time-series data. Data-points pertaining to every interval of 15 minutes are analyzed by *CloudPD* for anomalies.

6.4.2 Performance Models for Event Generation

We implemented three modeling techniques that can be used as part of the *Event Generation Engine* – Hidden Markov Models (HMM), nearest-neighbor (kNN), and k-means clustering [145]. All the three techniques attempt to qualitatively or quantitatively measure the extent of deviation of the current interval test data from models of normal behavior. The *CloudPD* architecture allows plug and play of any modeling technique as part of the *Event Generation Engine*. Here, we only discuss the kNN technique used in our evaluation, and refer the readers to [146] for details of the other two techniques.

This technique works by computing a *distance* measure between the data point under consideration and the nearest k neighbors in a given set of model data points known to be normal from recent history. In our implementation, the distance between two data points is defined as the sum of the differences between corresponding samples in the two data points (other reasonable definitions of the distance metric worked equally well). The kNN distance measure for a data point is the sum of the distances to its k nearest neighbors. Larger the distance, the larger is the deviation from the normal behavior. If the distance measure for the test interval's data points is higher by an empirically determined threshold, compared to the distances of the model data points, an event (alarm) is generated (for further analysis by the *Problem Determination Engine*). Note that, unlike HMM, kNN does not require any training and can learn the model of normal behavior in an online fashion, which allows it to adapt to changes in workload mix or intensity.

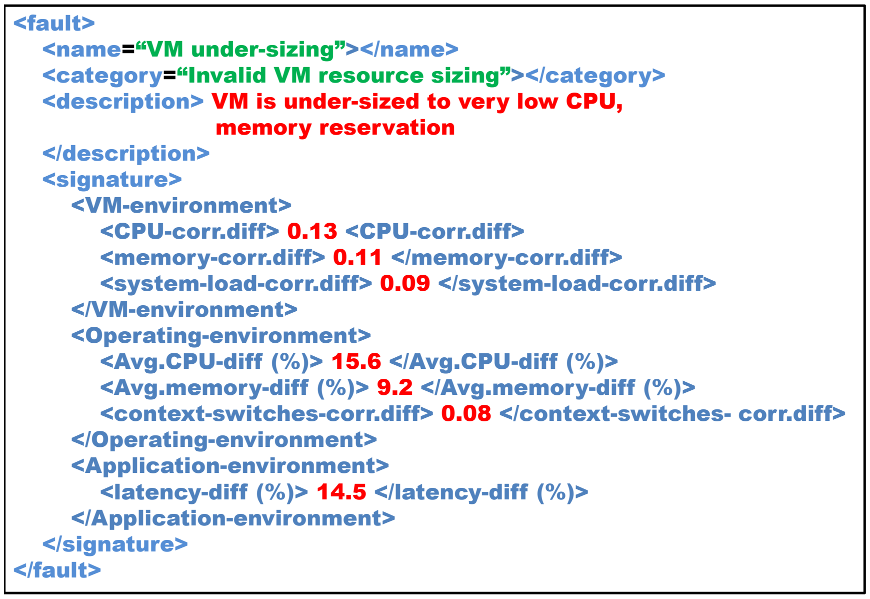
Table 6.1. System and application metrics monitored in *CloudPD*; Metrics part of operating context are marked with *.

System Metrics	**Description**	**Measurement Level**
cpu-user	% CPU time in user-space	$VM, Host^*$
cpu-system	% CPU time in kernel-space	$VM, Host^*$
memused	% of used memory	$VM, Host^*$
miss/s	# of cache misses per second	$Host^*$
ctxt	context switches per second	VM^*
eth-rxbyt	network bytes received per second	VM
eth-txbyt	network bytes transmitted per second	VM
pgpgin	KBytes paged-in from disk per second	VM
pgpgout	KBytes paged-out from disk per second	VM
fault	page faults per second	VM
system-load	processor's process queue length	VM
Application Metrics	**Description**	**Measurement Level**
Latency	response time	VM
Throughput	# of transactions/second	VM

6.4.3 Diagnosis Engine

We adopt an XML format for describing the fault signatures, so as to allow the *CloudPD* system to learn to classify new kinds of faults with expert assistance. An assumption that we make here is that each fault has a characteristic signature, which can be expressed in terms of the metrics monitored. We have observed this to be true in our extensive evaluations. We adopt a software wrapper based on Matlab *xmlwrite* utility to implement the diagnosis engine. Essentially, this wrapper provides two functionalities: (a) create a XML based signature of a new fault; and (b) compare a newly created signature with existing fault signatures, which are stored in a database.

Figure 6.5 provides an example of an expert-created signature for a VM



```
<fault>
   <name="“VM under-sizing”></name>
   <category="“Invalid VM resource sizing”></category>
   <description> VM is under-sized to very low CPU,
                 memory reservation
   </description>
   <signature>
      <VM-environment>
         <CPU-corr.diff> 0.13 <CPU-corr.diff>
         <memory-corr.diff> 0.11 </memory-corr.diff>
         <system-load-corr.diff> 0.09 </system-load-corr.diff>
      </VM-environment>
      <Operating-environment>
         <Avg.CPU-diff (%)> 15.6 </Avg.CPU-diff (%)>
         <Avg.memory-diff (%)> 9.2 </Avg.memory-diff (%)>
         <context-switches-corr.diff> 0.08 </context-switches- corr.diff>
      </Operating-environment>
      <Application-environment>
         <latency-diff (%)> 14.5 </latency-diff (%)>
      </Application-environment>
   </signature>
</fault>
```

Figure 6.5. Example signature of an invalid VM resizing fault.

resizing fault. The signatures are described using different contexts expressed as tags, namely, VM environment, operating environment, hypervisor (includes special log messages from the hypervisor), and application environment. Only the metrics that deviate from normal behavior are captured in the signature, and are expressed as thresholds denoting the minimum deviation in the correlation values required for a match with the signature. For example, the CPU correlation value computed between a pair of VMs hosting an application should deviate from the correlation value under normal behavior by at least as large as the defined threshold (*CPU-corr.diff*) to match the signature for a wrong VM sizing (likewise, other tag correlation differences also need to tally for a successful signature match).

6.4.4 Complexity Analysis

Let the number of VMs in a cluster executing the same application be N, the number of metrics monitored be M, and the number of fault types be F. Let T be the number of data points in each interval being analyzed. We analyze the complexity of each of the stages of *CloudPD*:

- Finding the nearest neighbor for each data point takes $O(T^2)$. Event generation using kNN performs this computation for each VM and each metric, which takes $O(NMT^2)$ time. This can be parallelized across $p \leq N$ threads, which would take $O(NMT^2/p)$ time.
- For each event generated, the problem determination stage performs $2(M+N)$ correlations (one each for the test interval data and the normal interval data obtained from recent history). Thus, net complexity is $O((M+N)T^2)$.
- Let M' be the maximum number of deviations in correlations that are part of the fault signature, and used to uniquely identify each fault. This is typically a small number (< 10). For each anomaly identified, diagnosis takes $O(FM')$ time.

Thus, the above analysis clearly suggests that the layered approach to fault diagnosis combined with the low complexity of each stage, allows *CloudPD* to learn fault models in an online manner for large-scale systems.

6.4.5 Types of Faults Handled by *CloudPD*.

Table 6.2 lists the various kinds of faults which we focus on in the design, implementation and evaluation of *CloudPD*.

Table 6.2. List of faults covered by *CloudPD*.

Cloud anomalies	Invalid VM resource sizing	Impact due to sharing	Invalid VM live migration	VM wrong reconfiguration
Non-cloud anomalies	Misconfigured application	Workload mix change	Workload intensity change	Software anomaly

6.4.6 Anomaly Remediation Manager

On the onset of anomalies, *CloudPD* detects, diagnoses and classifies them. It then suggests to the cloud manager, a bunch of ways to deal and recover from the faults. The particular set of preventive or remedial actions taken by *CloudPD* steady state management system are summarized in Table 6.3.

Table 6.3. Remedial actions taken by *CloudPD* on detecting various kinds of faults.

Fault	Handler	Action
Wrong VM Resizing	CloudRM	Trigger re-configuration
VM to VM Contention	CloudRM	Create collocation exclusion constraints between the VM trigger reconfiguration
Invalid VM Migration	CloudRM	Update the migration cost for the VM (to be used in future consolidations)
Application Error	System Admin	Open a service ticket
System Error	CloudRM, System Admin	Update server list. Trigger reconfigurations for fail-over. Raise problem ticket
Workload Intensity Change	CloudRM	Same as Wrong VM resizing
Workload Mix Change	CloudRM	Trigger re-profiling of the application. Reconfiguration, if needed by CloudRM

6.5 Experimental Evaluation

6.5.1 Experimental Setup

We used a virtualized server cluster to evaluate *CloudPD*. The server cluster consists of 1 IBM x3850 M2 server with 16 Xeon 2.132 GHz cores and 132 GB RAM, 3 HS 21 blade-servers with 4 Xeon 2.33 GHz cores and 8 GB RAM each, and 1 HS 21 blade-server with 4 Xeon 3 GHz cores and 10 GB RAM. The HS21 blades are hosted on an IBM Bladecenter-H chassis. The servers have a 2 Gbps Fiber Channel port, which is connected to an IBM DS4800 Storage Controller via a Cisco MDS Fiber Channel Switch. All servers run the VMWare ESXi 4 Enterprise Plus hypervisor and are managed by a VMWare vSphere 4 server. The cloud setup supports enterprise virtualization features including live VM migration, live VM resizing, VM cloning and VM snapshot.

We host total 28 VMs on our cloud setup, which run Ubuntu v10.04 64-bit operating system. Unless otherwise stated, all VMs are configured with 1.2 GB memory and 1.6 GHz CPU. Our VMs can be classified into three groups. The first group (*Hadoop cluster*) consists of 16 VMs that run Hadoop MapReduce [15].

The second (*Olio cluster*) consists of 6 VMs (4 VMs for web server and 2 VMs for database) and runs CloudStone [57], a multi-platform benchmark for Web 2.0. CloudStone consists of a load injection framework called Faban, and a social online calendar web application called Olio [57]. Faban is configured on a separate VM to drive the workload. The third (*RUBiS cluster*) runs the RUBiS e-commerce benchmark [127], and comprises of 6 VMs (2 VMs each across web, application and database tier).

6.5.2 Evaluation Metrics

We utilize the following four statistical measures to evaluate the effectiveness of anomaly detection and diagnosis by *CloudPD*. We define a successful anomaly detection as diagnosing the anomaly correctly using pre-defined fault signatures, along with localizing the affected VMs and metrics.

$$Recall = \frac{\text{Number of successful detections}}{\text{Total number of anomalies}} \tag{6.1}$$

$$Precision = \frac{\text{Number of successful detections}}{\text{Total number of alarms}} \tag{6.2}$$

$$Accuracy = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{6.3}$$

$$False\ Alarm\ Rate\ (FAR) = \frac{\text{Number of false alarms}}{\text{Total number of alarms}} = 1 - Precision \tag{6.4}$$

6.5.3 Competitive Methodologies

To the best of our knowledge, there is no prior work that performs an end-to-end detection, diagnosis and classification of faults in virtualized clouds. Hence, for the purpose of evaluation, we extend existing approaches to perform end-to-end problem determination in clouds. These schemes also implement a subset of our key design decisions, helping us understand the importance of each as employed by *CloudPD*. Hence, we evaluate the following four such schemes:

Baseline B1: This method is inspired from existing problem determination techniques that do not use operating context, such as [130, 103]. Hence, $B1$ uses only VM CPU and memory usage for problem determination. This scheme employs all other *CloudPD* techniques including the three-layered approach, usage of peers, and characterization of anomalies.
Baseline B2: It does not use the multi-layer approach and analyzes every interval in detail for anomalies. Hence, $B2$ also acts as an oracle and defines the boundaries for any correlation based technique for problem determination.
Baseline B3: It does not include the idea of correlation across peers. Hence, any gap between $B3$ and *CloudPD* can be attributed to the technique of correlating metrics across peers.
Baseline B4: It uses static thresholds to trigger the execution of the *Diagnosis Engine*, similar to monitoring systems like Ganglia and Nagios, and contrary to *CloudPD*'s online model-based dynamic thresholds. Hence, comparison of this method with *CloudPD* can highlight the importance of using an online learning approach for problem determination in clouds.

6.5.4 Stability of Correlation with Change in Workload and VM Configuration

We have used correlation across system metrics for a VM as well as correlation for a metric across peers to identify anomalies. Further, our problem signatures are also based on change in correlation between specific set of parameters. These problem signatures are defined independent of workload intensity, workload mix as well as VM configurations. Hence, our first set of experiments study if our assumption of correlation values being independent of these parameters is correct.

6.5.4.1 Stability of Correlation with Change in VM Configuration

In this experiment, we vary the configuration of VMs running the workloads. We configured a cluster of 8 Hadoop VMs ($VM_1 - VM_8$). The CPU and RAM reservations of $VM_1 - VM_4$ are set to the default values of 1.2 GHz and 1.6 GB,

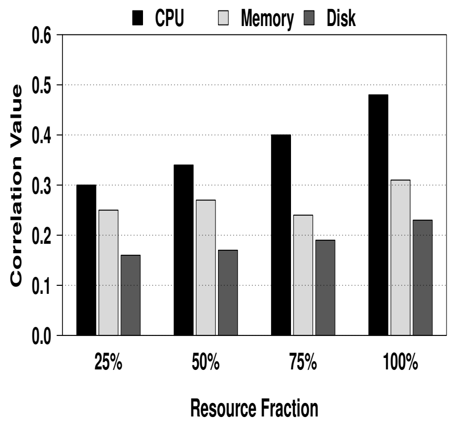
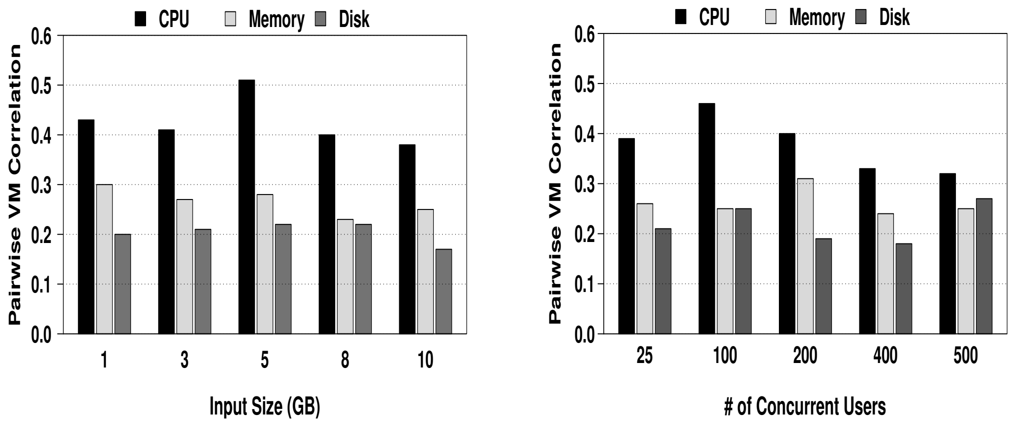


Figure 6.6. Generality of *CloudPD* in homogeneous and heterogeneous platforms.

respectively. Four experiments are conducted with different resource configurations for $VM_5 - VM_8$ (25%, 50%, 75% and 100% of the CPU and memory of $VM_1 - VM_4$). Hadoop `Sort` benchmark is run on 5 GB of data. We correlate CPU, memory and disk utilization across the VMs and report their average values in Figure 6.6. Although, the raw utilization might vary depending on the resource allocation to VMs, the correlation values remain nearly the same even for heterogeneous environments, demonstrating the stability of correlation values across VM configuration changes.

6.5.4.2 Stability of Correlation with Change in Workload Intensity

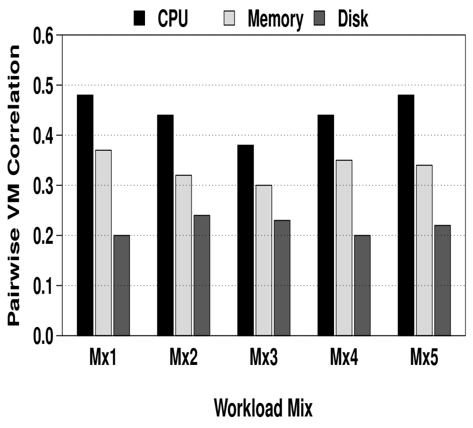


(a) Hadoop intensity (b) Olio intensity

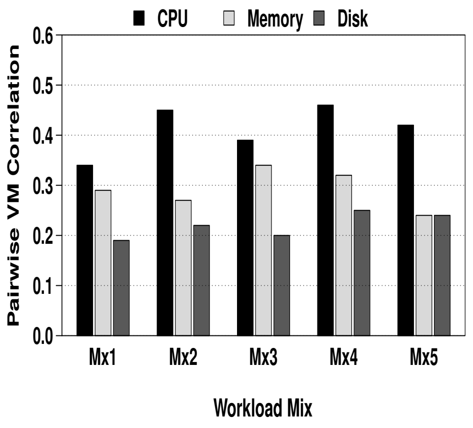
Figure 6.7. Stability of peer (VM) correlations across changes in workload intensity.

We next demonstrate the stability of *CloudPD* with changes in workload intensity. Figure 6.7(a) shows the variation in the average pairwise VM correlations (across CPU, memory and disk) with changes in workload intensity for Hadoop. The workload intensity change refers to running `Sort` with different input sizes from 1 to 10 GB. We observe that *CloudPD* will not trigger a fault for normal workload changes, since the pairwise VM correlations show very low variation across different workload intensities. As mentioned in Section 6.3.2.3, a fault affecting a specific VM will result in the correlation of the faulty VM with other VMs (for one or more metrics) to deviate from other pairwise correlations. Similar observation follows for the Olio benchmark as well, as shown in Figure 6.7(b). For Olio, change in workload intensity is achieved by changing the number of concurrent users from 25 to 500.

6.5.4.3 Stability of Correlation with Change in Workload Mix



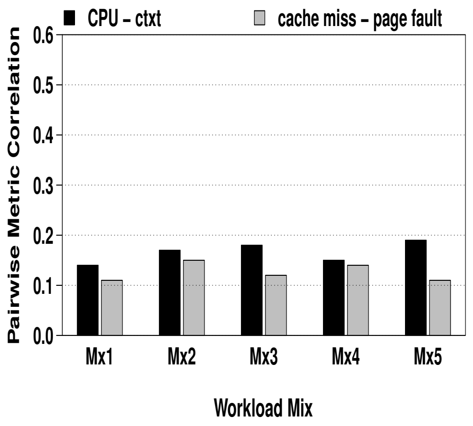
(a) Hadoop mix



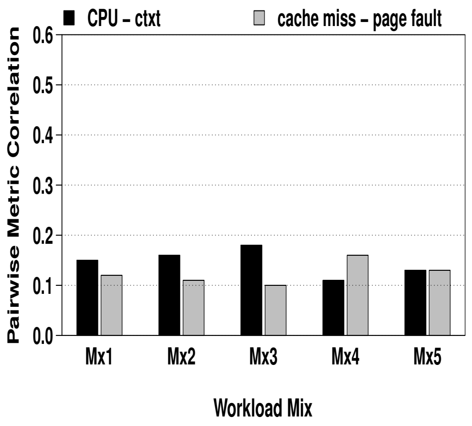
(b) Olio mix

Figure 6.8. Stability of peer (VM) correlations across changes in workload mix.

Changes in the transaction mix results in the resource usage behavior of the application to change, but correlation across all the VMs remains stable. The results of this experiment for Hadoop and Olio are shown in Figure 6.8, where we observe that the change in pairwise VM correlations is very low across different workload transaction mixes. The transaction mixes used are shown in Table 6.4. For Olio, $bm1-5$ corresponds to different operation mixes in terms of browsing



(a) Hadoop mix



(b) Olio mix

Figure 6.9. Stability of correlation across changes in workload mixes. $CPU - ctxt$ refers to correlation between CPU and context switches on the same VM. $cachemiss - pagefault$ implies correlation between cache misses and page faults.

transactions. Furthermore, the correlation across metric pairs on the same VM, although weaker, also remain stable across changes in workload mix, as depicted in Figure 6.9. A similar trend holds for cross-metric correlations on the same VM across workload intensity changes.

Table 6.4. Workload transaction mix specifications for Hadoop and Olio.

Mix-type	Hadoop	Olio
Mx1	(streamSort,javaSort)-(s)	bm1+ 80% rw-bd
Mx2	(streamSort,combiner)-(l)	bm2 + 85% rw-bd
Mx3	(webdataScan,combiner)-(m)	bm3 + 90% rw-bd
Mx4	(combiner,monsterQuery)-(l)	bm4 + 95% rw-bd
Mx5	(webdataScan,webdataSort)-(s)	bm5 + 99% rw-bd

Table 6.5. Number of injected faults (Synthetic Injection) or induced faults (Trace-driven).

Fault type	Synthetic expt.	Trace-driven expt.
invalid VM migration	3	4 (cloud induced)
invalid VM sizing	3	7 (cloud induced)
application misconfig.	4	0
CPU-hog	4	3
memory-hog	4	3
disk-hog	2	2
cache-hog	2	2
network-hog	2	2
Total number of faults	*24*	*23*

6.5.5 Synthetic Fault Injection Results

We compare the performance of various techniques in terms of their effectiveness in diagnosing faults, diagnosis time, and scalability. In these experiments, we ran each application (Hadoop, Olio or RUBiS) independently for 24 hours. For Hadoop, we ran the *Sort* benchmark on 5 GB data repeatedly. For Olio, we used a setting of 100 concurrent users and the default browsing transaction mix ($Mx1$ in Table 6.4). We used the default workload mix for RUBiS (read-only browsing mix and bidding mix with 15% read-write interactions). We divided each experiment into 96 intervals of 15 minutes each. *CloudPD* collects data from the *Monitoring Engine* and performs a diagnosis for each interval using all the competing methods. We inject faults randomly in 24 of these 96 intervals, the details of which are presented in Table 6.5. The resource-hog faults are custom-written pieces of code that continuously access a particular resource, mimicking a software anomaly. An interval is categorized as anomalous (for ground truth), if the application latency or throughput is deviant by a certain threshold (obtained through empirical analysis) from the latency/throughput observed for normal behavior. For Hadoop, the latency is the end-to-end job completion time; for RUBiS, latency is the end-to-end transaction response time. The latency threshold value chosen was 11%. For Olio, we used throughput, defined as the number of transactions per second, as the application performance metric, and the threshold for categorizing an interval as anomalous was set to 9%. Here, we only present results for Hadoop and Olio, but similar results follow for RUBiS and refer readers to [146] for RUBiS specific results.

6.5.5.1 End-to-end Diagnosis Comparison

Table 6.6 shows the end-to-end diagnosis results for *CloudPD* and the competing methods. For Hadoop, *CloudPD* was able to correctly detect and diagnose 21 out of the 24 faults and 69 out of the 72 normal intervals. It compares very favorably with Oracle $B2$, which is able to identify only one more additional anomaly compared to *CloudPD* with exhaustive analysis. $B1$ has low recall and precision as it monitors only a VM's CPU and memory, and ignores other system metrics (both at VM and

Table 6.6. Comparing End-to-end Diagnosis Effectiveness for Hadoop, Olio and RUBiS.

Method	**# of correct normal detections**	**# of correct anomalous detections**	**# of correct Phase 1**	**# of total predicted anomalies**	**Recall**	**Precision**	**Accuracy**	**FAR**
CloudPD	69	21	23	24	0.88	0.88	0.88	0.12
B1	62	11	15	21	0.46	0.52	0.49	0.48
B2	69	22	24	25	0.92	0.88	0.90	0.12
B3	64	12	23	20	0.50	0.60	0.54	0.40
B4	66	15	15	21	0.63	0.71	0.67	0.29
Hadoop								
CloudPD	68	20	22	24	0.83	0.83	0.83	0.17
B1	62	11	15	21	0.46	0.52	0.49	0.48
B2	68	22	24	26	0.92	0.85	0.88	0.15
B3	63	11	22	20	0.46	0.55	0.50	0.45
B4	63	13	14	22	0.54	0.59	0.56	0.41
Olio								
CloudPD	68	20	22	24	0.83	0.83	0.83	0.17
B1	61	12	15	23	0.50	0.52	0.51	0.48
B2	68	22	24	26	0.92	0.85	0.88	0.15
B3	62	12	22	22	0.50	0.54	0.52	0.46
B4	63	14	13	23	0.59	0.61	0.60	0.39
RUBiS								

server level). Further, it also has a high false alarm rate, where change in operating context is classified as an anomaly. *CloudPD* is able to avoid these false alarms as it correlates data across multiple metrics (eg., CPU with context switches and memory with page faults) and does not report an anomaly if the correlations are consistent with the learning models. $B3$ also recorded a low recall and precision and a high false alarm rate, as it does not correlate across peers (VMs running the same application). However, its performance is better than $B1$. $B4$, that uses static thresholds again does not have satisfactory performance. Similar results are observed for Olio and RUBiS, although the performance of *CloudPD* is slightly poorer compared to Hadoop. We conjecture that this is because Hadoop is a symmetric batch application (map/reduce tasks are similar in type and intensity across all VMs with time), whereas Olio is a more generic distributed application with greater burstiness. There exists intra- and inter-tier correlations among VMs for Olio. However, the magnitude of intra-tier VM correlations is higher than inter-tier VMs correlations, which results in some error in detecting anomalies. We emphasize the fact that *CloudPD* is effective in accurately distinguishing cloud anomalies from workload changes and application faults even for shared resources such as cache and disk.

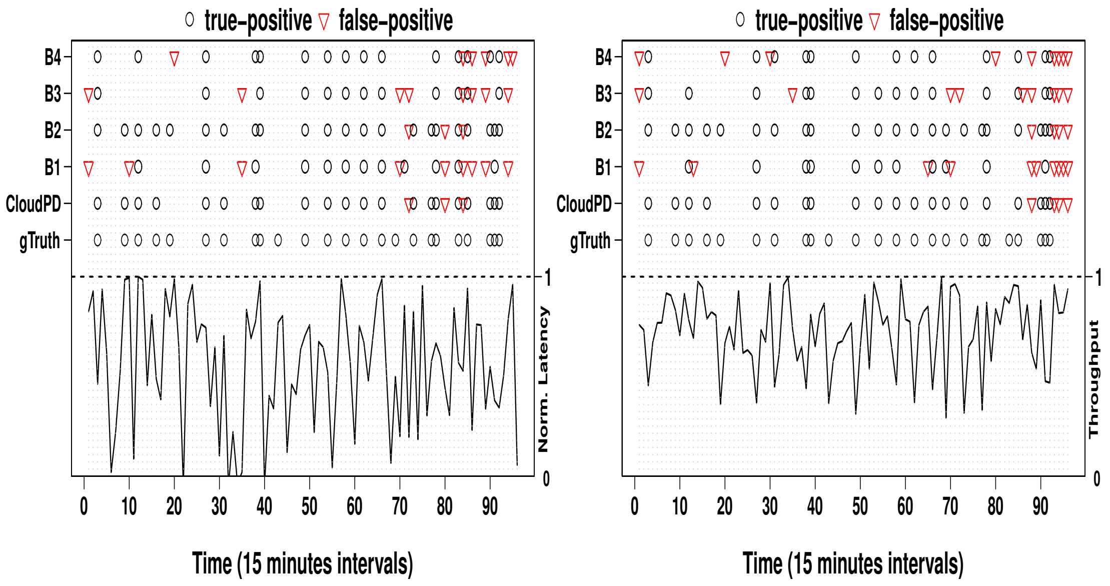
Table 6.7. Undetected anomalies for Hadoop.

Method	**Undetected anomalies**
CloudPD	1 disk hog + 2 application misconfig. (total 3)
B1	2 network hog + 2 disk hog + 2 cache hog + 2 application misconfig. + 2 invalid VM sizing + 3 invalid VM migration (total 13)
B2	2 application misconfig. (total 2)
B3	2 disk hog + 2 cache hog + 2 application misconfig. + 2 memory hog + 1 CPU hog + 3 invalid VM migration (total 12)
B4	2 disk hog + 1 cache hog + 1 memory hog + 2 application misconfig. + 3 invalid VM migration (total 9)

Table 6.8. Undetected anomalies for Olio.

Method	**Undetected anomalies**
CloudPD	2 disk hog + 2 application misconfig. (total 4)
B1	2 network hog + 2 disk hog + 2 cache hog + 2 application misconfig. + 2 invalid VM sizing + 3 invalid VM migration (total 13)
B2	2 application misconfig. (total 2)
B3	2 disk hog + 2 cache hog + 3 application misconfig. + 2 memory hog + 1 CPU hog + 3 invalid VM migration (total 13)
B4	2 disk hog + 1 cache hog + 2 memory hog + 3 application misconfig. + 3 invalid VM migration (total 11)

We further analyze the specific faults that were undetected by *CloudPD* and the four baselines. These are listed in Tables 6.7 and 6.8, for Hadoop and Olio, respectively. *CloudPD* failed to identify 1 disk hog and 2 application misconfiguration faults. The reason disk hog eluded detection can be explained by the fact that the VMs share their local disks across a Storage Area Network (SAN). The deviation of disk utilization from normal values was not sufficient for the *Event Generation Engine* to raise an alarm, preventing *CloudPD* from detecting it. However, $B2$ was able to detect this as correlations across metrics and VMs calculated by the *Problem Determination Engine* showed a deviation from normal behavior. *CloudPD* could not identify 2 application misconfiguration faults as well, as the difference in cross-resource and cross-VM correlation values from normal was not high enough to mark them as anomalies. $B2$ missed the same 2 application misconfiguration faults as *CloudPD*. As $B1$ only monitors CPU and memory, it missed a total of 13 faults (it was effective in identifying only CPU and memory hog faults). $B3$ failed to detect most application related faults as it only performs correlations across resources within a VM and does not perform cross-VM correlations. A faulty VM considered by itself appears as though it is servicing a very large workload, and hence was not tagged as anomalous. $B4$ is sensitive to the specific thresholds that are set, and is



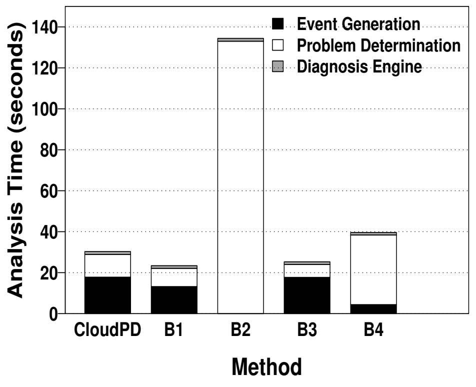
(a) Hadoop (b) Olio

Figure 6.10. Time-series showing the effectiveness of *CloudPD* and other base methods in detecting faults across Hadoop and Olio bechmarking systems.

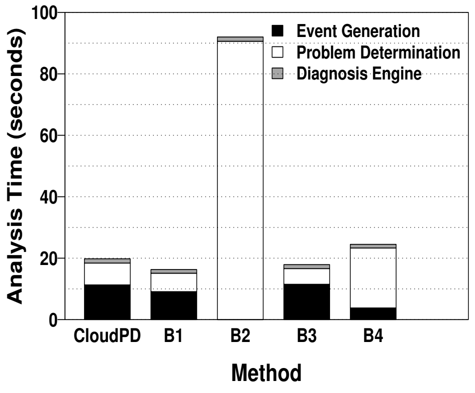
ineffective in identifying different manifestations of the same type of fault. Figure 6.10 shows the run-time behavior of system in the presence of faults, and how *CloudPD* and other baselines perform in terms of identifying or missing the injected faults. This holds across both Hadoop and Olio benchmarks.

6.5.5.2 Diagnosis Time and Scalability

Effective diagnosis is just one of the goals for problem determination in cloud. A dynamic and autonomic system like cloud requires diagnosis to be performed quickly for efficient dispatch of remediation actions. Figure 6.11 shows the calibration and analysis time of each stage for *CloudPD* and other base methods with Hadoop and Olio benchmarks. The numbers are averaged across the 96 intervals in the 24-hour experiment. Note that the system state can change every 15 minutes and hence remediation is relevant only if performed in time much less than 15 minutes. The *Event Generation* of *CloudPD* takes on an average 17.8 seconds for Hadoop and is executed for every interval, and on data for every VM and metric monitored. The *Problem Determination* is triggered only if an alarm is raised by the *Event Generation*. For other intervals, the time taken by *Problem Determination* is taken as zero, while computing the average. Hence, although *Problem Determination* takes longer than *Event Generation* (about 40s for Hadoop), since the latter is invoked only selectively, the analysis time is lower. The same is true for *Diagnosis Engine* as it is invoked only if an anomaly is detected, allowing *CloudPD* to quickly detect faults (time taken only marginally higher than $B1$ and $B3$, which detect very few anomalies). In comparison, Oracle $B2$ has no *Event Generation*, and hence the time spent in *Problem Determination* is $10X$ larger than *CloudPD*.



(a) Hadoop analysis time

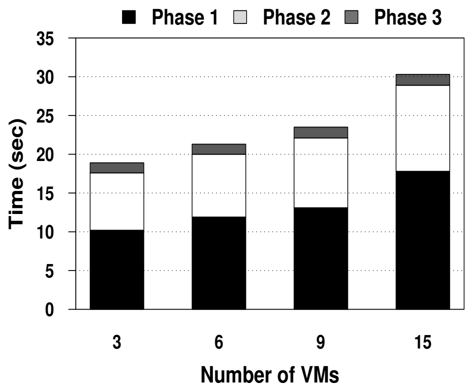


(b) Olio analysis time

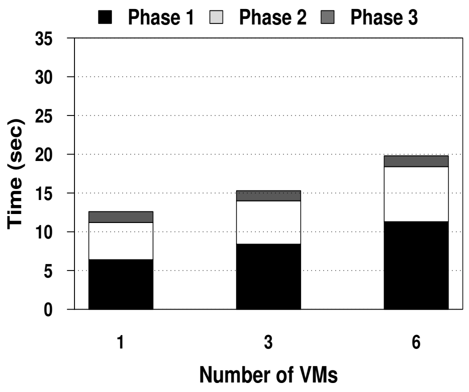
Figure 6.11. Analysis time of each stage across *CloudPD* and base methods.

The time spent analyzing the Olio cluster is lesser compared to Hadoop as the cluster size is smaller. Note that *CloudPD* spends most of the time in *Event Generation* which is completely parallelizable and can be easily reduced further.

Figure 6.12 shows the effect of increase in the number of VMs hosting an application on the calibration and analysis time of *CloudPD*. The analysis time is shown as histograms with breakup of time taken by each stage of analysis for different number of VMs in the Hadoop and Olio clusters. We notice that the analysis time grows sub-linearly with the number of VMs, with the increase in analysis time being mainly in the first event generation phase. Further, as noted before, event generation time can be reduced by parallelizing it across different VMs. This experiment establishes the effectiveness of *CloudPD*'s multi-layer approach, allowing it to scale to larger systems with more VMs and servers.



(a) Hadoop VM scaling



(b) Olio VM scaling

Figure 6.12. Effect of VM scaling on calibration and analysis time of *CloudPD*.

Table 6.9. Comparing end-to-end diagnosis effectiveness for trace-driven study.

Method	# of correct normal detections	# of correct anomalous detections	# of correct Phase1	# of total predicted anomalies	Recall	Precision	Accuracy	FAR
CloudPD	67	18	21	24	0.78	0.75	0.77	0.25
B1	58	10	14	25	0.43	0.40	0.42	0.60
B2	67	21	23	27	0.91	0.78	0.84	0.22
B3	60	11	21	24	0.48	0.46	0.47	0.54
B4	60	13	15	26	0.57	0.50	0.53	0.50

6.5.6 Trace-driven Fault Generation Results

Our next set of experiments were conducted to study the effectiveness of *CloudPD* in a real cloud setting, where cloud configuration actions spontaneously generate faults. Our cloud testbed is managed by a cloud management stack that could provision virtual machines and perform dynamic consolidation to reduce power. We used the *pMapper* consolidation manager [137], which has been studied by multiple researchers and productized. The consolidation involves reconfiguration of VM resource allocations as well as changing the placement of VMs on physical servers every 2 hours. The cloud testbed hosts a Hadoop cluster consisting of 10 VMs and an Olio cluster consisting of 6 VMs, with 4 VMs for the web server tier, and 2 VMs for the database tier.

We used two real traces from a production data center of a Fortune 500 company running enterprise applications to drive the workload for Hadoop and Olio. The two traces contain a time-series of CPU and memory utilization across the servers, with a data point every 15 minutes for a total duration of 24 hours, which is also the duration of our experiment. We built a profile of Hadoop that captured the relationship of workload size with CPU and memory utilization. We similarly created a profile of Olio to capture the relationship of number of concurrent users with CPU and memory utilization of the Web and database tier. In a given interval, we ran Hadoop with a data size that generates the CPU utilization provided by the trace for that interval. Similarly, we ran Olio with number of users such that the utilization of the web server matched what was specified by the trace for that interval. Hence, this experiment captures changes in workload intensity that happen in real clouds. The resource profiles are captured by the following equation obtained through linear regression:

$$\begin{aligned} CPU(Olio-Web) &= Users * 0.1 + 5 \\ CPU(Olio-DB) &= Users * 0.035 + 7.5 \\ CPU(Hadoop) &= DataSize * 2.83 + 12.9 \end{aligned} \tag{6.5}$$

CloudPD independently monitors the cluster and identifies cloud related anomalies, workload intensity and workload mix changes, as well as application anomalies for 15 minute intervals. We randomly injected anomalies in some of these intervals as listed in Table 6.5. Apart from the injected application anomalies, we noticed that 4 intervals experienced invalid VM migrations and 7 intervals experienced invalid VM sizing anomalies due to cloud reconfiguration (we did not inject any cloud related anomalies). These were determined to be anomalous as the application latency and throughput deviated by 11% and 9% for Hadoop and Olio, respectively. The sizing faults were a result of prediction error and the live migration faults were due to high resource utilization on the servers.

Figure 6.13 provides a detailed view of the case-study. For each of the 96 intervals (shown in the X-axis), the left Y_1-axis shows the ground truth of intervals that had anomalies along with anomaly predictions made by *CloudPD* and the four baselines. Both correct anomalous detections and false positives are marked. On the Y_2-axis, we plot the normalized application latency, where a value of 1 denotes the interval with the highest average job latency. Observe that anomalous intervals (marked under $gTruth$) have a higher latency than normal intervals. One can observe that $B1$, $B3$ and $B4$ have many false positives, which is consistent with our studies using synthetic fault injection (Section 6.5.5).

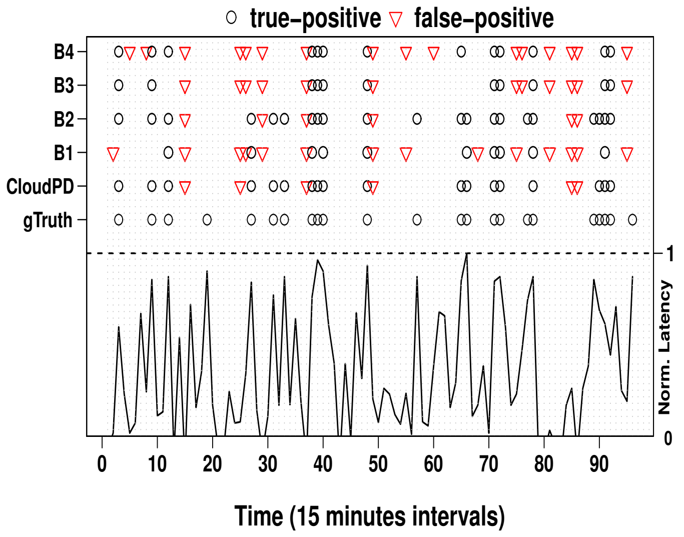


Figure 6.13. Time-series showing the effectiveness of *CloudPD* and other base methods in detecting faults in a 24-hour enterprise trace-based case-study.

The performance metrics for *CloudPD* and the baselines in terms of recall, precision, and other metrics are summarized in Table 6.9. *CloudPD* is able to correctly diagnose 18 out of the 23 anomalous intervals and identify 67 out of the 73 normal intervals as normal. This is close to the performance of baseline $B2$ and significantly better than the other three baselines. This ability of *CloudPD* is due to its unique characteristic to better manage and deal with shared resources such as disk and cache in segregating cloud anomalies from application faults and normal workload change. Table 6.10 provides the list of anomalies that were undetected by *CloudPD* and the baselines. *CloudPD* fails to detect the 2 disk hog anomalies, the reason for which can be attributed to the fact that the VMs share a SAN storage and the deviation of the storage utilization values from normal was not significant enough for the event generation phase of *CloudPD* to raise an alarm. *CloudPD* also missed detecting 2 invalid resizing events and an invalid VM migration. These were marginal anomalies which caused the application latency to be high enough to be classified as an anomaly, but the correlation values were not deviant enough for *CloudPD* to detect them.

Table 6.10. Undetected anomalies for case-study.

Method	Undetected anomalies
CloudPD	2 disk hog + 2 invalid VM sizing + 1 invalid VM migration (total 5)
B1	2 network hog + 2 disk hog + 2 cache hog + 3 invalid VM sizing + 4 invalid VM migration (total 13)
B2	1 disk hog + 1 invalid VM sizing (total 2)
B3	2 disk hog + 2 cache hog + 2 memory hog + 1 CPU hog + 2 invalid VM sizing + 3 invalid VM migration (total 12)
B4	2 disk hog + 2 cache hog + 1 memory hog + 1 CPU hog + 2 invalid VM sizing + 2 invalid VM migration (total 10)

Table 6.11. CPU usage (% CPU time), memory usage and network bandwidth overhead during the data collection using *sar* utility.

	% CPU	Memory (MB)	Network BW (KB/s)
Hadoop	0.35	0.80	37.8
Olio	0.18	0.44	14.5
Case study	0.41	0.92	40

6.5.7 Diagnosis Overheads

CloudPD uses cloud resources for diagnosing faults. We quantify the overhead of *CloudPD* in terms of the CPU, memory and network bandwidth usage for our experiments with synthetic faults as well as the trace-driven study. We report the resource utilization averaged across VMs and over the duration of the experiment (24 hours) in Table 6.11. We observe that *CloudPD* introduces minimal overhead on the system. Our experimental study, thus, establishes the effectiveness of *CloudPD* to accurately diagnose application, system, and cloud-induced faults quickly, with low time and system overhead.

6.6 Chapter Summary

In this chapter, we proposed a light-weight, automated, and accurate fault detection and diagnosis system called *CloudPD* for shared utility clouds. *CloudPD* uses a layered online learning approach to deal with the frequent reconfiguration and higher propensity of faults in clouds. We introduced the notion of operating context, which is essential to identify faults that arise due to the shared nature of non-virtualized resources. *CloudPD* monitors a wide range of metrics on the VMs as well as the physical servers, compared to only CPU and memory metrics monitored by most prior work in the area. *CloudPD* diagnoses anomalies based on pre-computed fault signatures and allows remediation to be integrated with cloud steady state management in an automated fashion. Using cloud representative workloads like Hadoop, Olio and RUBiS, we demonstrate the effectiveness of *HybridMR* in detecting and diagnosing anomalies with low false positives and high accuracy of 88%, 83% and 83%, respectively. Moreover, in an enterprise trace-based case study, *HybridMR* diagnosed problems within 30 seconds with an accuracy of 77%.

Chapter 7

Conclusions and Future Research Directions

7.1 Conclusions

The era of cloud computing is here, with more and more enterprise services moving to cloud platforms to achieve better economy of scale, elasticity and flexibility. With the wide spread emergence of different flavors of cloud computing, where both cloud providers and end users leverage the promises and benefits, the thrust of providing the required guarantees towards predictable performance and reliability is becoming essential. This dissertation contributes towards this goal of in-depth investigation of some of the critical performance and reliability challenges, and proposes novel methodologies, structured in the following four parts:

In the first part (Chapter 3), we investigated the characterization and modeling of cloud workloads. We focused on an important characteristic of cloud workloads, called *task placement constraints*, which are predicates of a task's preference for certain machine properties. Specifically, we contributed by modeling and synthesizing task placement constraints in large compute clusters like the ones in Google cloud back-end. The key insight here is to capture the scheduling performance impact of task placement constraints. We observed anywhere between 2 to 6 times increase in task scheduling delays due to the presence of task constraints, which means tens of minutes of additional task wait times. We proposed a technique to quantify the relative performance impact of the constraints. Finally, we developed benchmarking algorithms to augment existing performance benchmarks with task constraints and machine properties, to make them more realistic and representative of cloud workloads. Results indicate that our benchmarking algorithms can reproduce the performance aspects of Google compute clusters with respect to task scheduling delay and resource utilization with reasonable accuracy.

The design and implementation of an efficient resource management framework for Hadoop MapReduce clusters is investigated in the second part (Chapter 4). Currently, Hadoop MapReduce framework suffers from the fixed-size, static and uncoordinated slot-based resource scheduling schemes. The resulting liabilities are prolonged job completion time and reduced cluster resource utilization. Motivated by the need to eliminate the slot-based performance bottlenecks in Hadoop, we proposed *MROrchestrator*, a flexible and efficient resource allocation framework for Hadoop MapReduce. *MROrchestrator* addresses the shortcomings prevalent in current Hadoop MapReduce in terms of resource scheduling. The main idea here is to provide a fine-grained, dynamic and coordinated resource management framework, that is aware of the run-time resource profiles of constituent map/reduce tasks, while performing scheduling of shared resources. Through extensive evaluations with representative MapReduce workloads suite, we demonstrated the benefits of *MROrchestrator* in terms of reducing the job completion time by up to 38% and providing a 25% improvement in cluster resource utilization.

In the third part (Chapter 5), we describe a 2-phase hierarchical scheduler, called *HybridMR*, for hybrid data centers, consisting of a mix of native and virtual machines, to leverage the benefits of both paradigms. In the first phase, *HybridMR* profiles incoming MapReduce jobs to determine the estimated virtualization overheads, and uses this information to automatically guide placement between physical machines and virtual machines. In the second phase, *HybridMR* builds run-time resource prediction models, and performs dynamic resource orchestration to minimize the interference within and across collocated MapReduce and interactive applications. Detailed evaluations on a hybrid cluster consisting of 24 physical servers and 48 virtual machines, with diverse workload mix of interactive and batch MapReduce applications demonstrate that *HybridMR* achieves up to 40% improvement in job completion time of MapReduce jobs over a virtual Hadoop, while satisfying the SLAs of interactive applications. Further, *HybridMR* provides 45% improvement in resource utilization and around 43% energy savings compared to a native Hadoop with minimal performance penalty. Moreover, we demonstrate that it is possible to dynamically change the native and virtual cluster configurations to accommodate variations in workload mix for maximizing the performance-energy envelope. These results suggest that a hybrid cluster configuration is a better cost-effective solution than the either-or approach to native and virtual modes of computation.

In the fourth part (Chapter 6), we present the design and implementation of a novel end-to-end problem detection and diagnosis framework for virtualized cloud platforms, called *CloudPD*. *CloudPD* is a light-weight, automated, and accurate fault detection and diagnosis system for shared utility clouds. It uses a layered online learning approach to deal with the frequent reconfiguration and higher propensity of faults in clouds. We introduced the notion of operating context, which is essential to identify faults that arise due to the shared nature of non-virtualized resources. *CloudPD* monitors a wide range of metrics on the VMs as well as the physical servers, compared to only CPU and memory metrics monitored by most prior work in the area. *CloudPD* diagnoses anomalies based on pre-computed fault signatures and allows remediation to be integrated with cloud steady state management in an automated fashion. Using a prototype implementation with cloud representative workloads like Hadoop, Olio and RUBiS, we demonstrate that *HybridMR* detects and diagnoses faults with low false positives and high accuracy of 88%, 83% and 83%, respectively. In an enterprise trace-based case study, *HybridMR* diagnosed problems within 30 seconds with an accuracy of 77%, demonstrating its effectiveness in real-life operations. To the best of our knowledge, *CloudPD* is the first end-to-end fault management system that can detect, diagnose, classify and suggest remediation actions for virtualized cloud-based anomalies.

7.2 Future Research Directions

Each of the four proposals described in this dissertation has prospective avenues and scope for future research extensions. Some of the specific future research directions in the context of each are described below:

7.2.1 Future Work on Cloud Workload Characterization

- It would be interesting to investigate the impact of constraints that are preferences rather than requirements. For example, in some situations, a task may prefer to run on a machine with 4 cores, but the task may not require this. We expect that *Utilization Multiplier* metric will be a useful tool in these studies since it allows us to understand effective task utilization with and without satisfying a preferential constraint.
- We plan to analyze the group of constraints that apply to a collection of tasks. For example, there may be a requirement that tasks be assigned to the same machine because of shared data. Characterizing and benchmarking with inter-task constraints is complicated because tasks and machines cannot be addressed in isolation.
- With the increasing level of cloud heterogeneity, optimal scheduling of resources to jobs from multiple tenants require effective synergy of application demands and supply of cloud resources [147]. For example, a job may require a specific type and amount of a resource instance, a second job may benefit from having access to a GPU and small core machine, while the third might run smoothly if it is not collocated with another job from another user. Deriving insights from the performance implications of task placement constraints, we plan to develop new scheduling algorithms that explicitly leverage the demand of tasks for specific constraints and the supply of machines with matching properties. We are interested to explore another category of constraints that relates to fault-tolerance, cluster-level properties and hard constraints (ones which are requirements rather than preferences) [147, 148]. A scheduler that explicitly recognizes this cluster

and application level heterogeneity via constraints specifications can do a better errand of job scheduling and resource assignment. As a continuation, we also plan to develop a simulation platform to conduct such scheduling/QoS/performance trade-off studies. Another related focus is to develop an initial version of a mathematical (and/or control theoretic) scheduling framework that explicitly incorporates the notion of task placement constraints and machine properties.

- We look forward to seeing data that provides insights into the performance impact of task placement constraints in other cloud representative clusters. We plan to evaluate the generality of our proposed methodology of workload characterization and derive useful performance insights. We will be using the recently released one month data from a Google cluster consisting of eleven thousand machines [47] for these future studies.

7.2.2 Future Work on Resource Management in Hadoop MapReduce

- We plan to extend *MROrchestrator* for controlling other shared resources like disk and network bandwidth, besides CPU and memory.
- We look forward to design a resource scheduling discipline for Hadoop MapReduce clusters that (i) augments the resource management framework, *MROrchestrator* for better resource scheduling; (ii) harnesses the heterogeneity of the underlying infrastructure. We plan to leverage here the task placement constraint ideas (Chapter 3) to better match the demands/preferences of tasks for certain machine configurations with available machine properties, besides their resource requirements.
- The large-scale utility of *MROrchestrator* can be demonstrated by evaluating it on a bigger experimental environments and public clouds like Amazon EC2.

7.2.3 Future Work on Workload Scheduling in Hybrid Data Centers

- We plan to extend *HybridMR* by making the 2-phase scheduler aware of the data locality design metric. This requires developing efficient locality-aware scheduling algorithms, for placement of MapReduce jobs.
- There are opportunities to develop generic interference models for resources like CPU, memory, I/O and network, is critical towards understanding the run-time performance of collocated interactive and batch jobs. We have developed an initial resource interference performance model for CPU and memory for MapReduce jobs. We plan to build upon this to design models across all critical resources and include both interactive and batch applications.
- We look forward to extend this study of hierarchical scheduling in the context of optimizing MapReduce on shared memory multi-core platforms. Specifically, we plan to explore the following two research directions: (i) analyze the network traffic characteristics in the context of running MapReduce framework on shared memory multi-core systems. This is because with the emergence of on-chip networks as the scalable communication fabric in multi-core systems including system-on-chips (SoCs) and chip multiprocessors (CMPs), it is becoming increasingly important to characterize and model the spatial and temporal variations of traffic to better understand the various performance and power implications of running applications in these environments; (ii) leverage important results and insights from the characterization of Network-on-Chips (NoCs) to better inform the MapReduce scheduler while scheduling the constituent map/reduce tasks of jobs, with an objective to minimize the network communication overheads, and hence improve the performance of hosted MapReduce jobs.
- It would be interesting to evaluate *HybridMR* on a larger scale hybrid data center with more number of native and virtual machines to assess and demonstrate the scalability aspect.
- To get an estimate of the scheduling optimality, an in-depth boundary analysis based on a theoretical model, would be interesting.

7.2.4 Future Work on Fault Diagnosis in Clouds

- We plan to extend *CloudPD* with more diverse and extensive types of anomalies, with their associated fault signatures. There is scope to fine-tune the Diagnosis Engine by creating a more detailed and across-the-board database of known fault signatures.
- An important next step is to analyze and investigate new set of performance anomalies that are the manifestations of virtualization artifacts.
- We look forward to evaluate *CloudPD* in typical large cloud environments like IBM Smart Cloud [149] and Amazon EC2 [1].
- There is scope to integrate *CloudPD* with a dynamic cloud reconfiguration manager, with the objective to provide an end-to-end dynamic, flexible, and robust fault detection and diagnosis framework for virtualized cloud systems.

To conclude, while there are many promises and opportunities that the cloud computing paradigm heralds, there are associated challenges and concerns. Amongst them, performance guarantees and reliability assurance are of foremost importance. With this motivation, this dissertation contributes towards improving the performance and reliability issues in cloud platforms. Through effective workload characterization, resource management, workload scheduling and fault diagnosis, this dissertation makes sincere attempts to extensively investigate the proposed research aspects, and pursue novel avenues for future research explorations.

Bibliography

[1] "Amazon Elastic Compute Cloud," `http://aws.amazon.com/ec2`.

[2] "Google App Engine," `http://code.google.com/appengine`.

[3] Verma, A., G. Dasgupta, T. K. Nayak, P. De, and R. Kothari (2009) "Server Workload Analysis for Power Minimization using Consolidation," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, USENIX Association, pp. 28–28.

[4] Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia (2010) "A View of Cloud Computing," *Commun. ACM*, **53**(4), pp. 50–58.

[5] Cecchet, E., V. Udayabhanu, T. Wood, and P. Shenoy (2011) "BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications," in *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, USENIX Association, pp. 4–4.

[6] Mishra, A. K., J. L. Hellerstein, W. Cirne, and C. R. Das (2010) "Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters," *SIGMETRICS Perform. Eval. Rev.*, **37**(4), pp. 34–41.

[7] Zhang, Q., J. L. Hellerstein, and R. Boutaba (2011) "Characterizing Task Usage Shapes in Google's Compute Clusters," in *Large Scale Distributed Systems and Middleware Workshop, LADIS 2011*.

[8] Reiss, C., A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch (2012) "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, ACM, pp. 7:1–7:13.

[9] Condor Project, "University of Wisconsin," `http://www.cs.wisc.edu/condor`.

[10] Thain, D., T. Tannenbaum, and M. Livny (2005) "Distributed Computing in Practice: The Condor Experience: Research Articles," *Concurr. Comput. : Pract. Exper.*, **17**(2-4), pp. 323–356.

[11] "IBM LoadLeveler," `http://tinyurl.com/ibmloadleveller`.

[12] Zhou, S., X. Zheng, J. Wang, and P. Delisle (1993) "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *Softw. Pract. Exper.*, **23**(12), pp. 1305–1336.

[13] Foster, I. and C. Kesselman (1996) "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputer Applications*, **11**, pp. 115–128.

[14] Dean, J. and S. Ghemawat (2008) "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, **51**(1), pp. 107–113.

[15] Apache, "Hadoop," `http://hadoop.apache.org`.

[16] "Hadoop MapReduce," `http://tinyurl.com/Hadoop-MR`.

[17] Ghodsi, A., M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica (2011) "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, USENIX Association, pp. 24–24.

[18] Ananthanarayanan, G., S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris (2010) "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, USENIX Association, pp. 1–16.

[19] Gallet, M., N. Yigitbasi, B. Javadi, D. Kondo, A. Iosup, and D. Epema (2010) "A Model for Space-Correlated Failures in Large-Scale Distributed Systems," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, Springer-Verlag, pp. 88–100.

[20] Benson, T., S. Sahu, A. Akella, and A. Shaikh (2010) "A First Look at Problems in the Cloud," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, USENIX Association, pp. 15–15.

[21] Compuware.com (2011), "Performance in the Clouds," White paper.

[22] "Cloud Computing Models," `http://tinyurl.com/cloud-models`.

[23] "Google Compute Engine," `https://cloud.google.com/products/compute-engine`.

[24] "Microsoft Windows Azure," `www.windowsazure.com`.

[25] "Rackspace Open Cloud," `www.rackspace.com`.

[26] Lin, J. and C. Dyer (2010) *Data-Intensive Text Processing with MapReduce*, Synthesis Lectures on Human Language Technologies, Morgan and Claypool Publishers.

[27] "Big Data," `http://en.wikipedia.org/wiki/Big_data`.

[28] Amazon, "AWS," `http://aws.amazon.com`.

[29] "Amazon MapReduce," `http://aws.amazon.com/elasticmapreduce`.

[30] Barford, P. and M. Crovella (1998) "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, ACM, pp. 151–160.

[31] Arlitt, M. F. and C. L. Williamson (1996) "Web Server Workload Characterization: The Search for Invariants," in *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '96, ACM, pp. 126–137.

[32] Arlitt, M. and T. Jin (1999) *Workload Characterization of the 1998 World Cup Web Site*, *Tech. rep.*, IEEE Network.

[33] Chapin, S. J., W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby (1999) "Benchmarks and Standards for the Evaluation of Parallel Job Schedulers," in *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '99/JSSPP '99, Springer-Verlag, pp. 67–90.

[34] Downey, A. B. and D. G. Feitelson (1999) "The Elusive Goal of Workload Characterization," *SIGMETRICS Perform. Eval. Rev.*, **26**(4), pp. 14–29.

[35] Cirne, W. and F. Berman (2001) "A Comprehensive Model of the Supercomputer Workload," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, IEEE Computer Society, pp. 140–148.

[36] Lublin, U. and D. G. Feitelson (2003) "The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs," *J. Parallel Distrib. Comput.*, **63**(11), pp. 1105–1122.

[37] Ersoz, D., M. S. Yousif, and C. R. Das (2007) "Characterizing Network Traffic in a Cluster-based, Multi-tier Data Center," in *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, IEEE Computer Society, pp. 59–59.

[38] Schroeder, B., E. Pinheiro, and W.-D. Weber (2009) "DRAM Errors in the Wild: A Large-Scale Field Study," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, ACM, pp. 193–204.

[39] Leung, A. W., S. Pasupathy, G. Goodson, and E. L. Miller (2008) "Measurement and Analysis of Large-Scale Network File System Workloads," in *Proceedings of USENIX Annual Technical Conference*, ATC'08, USENIX Association, pp. 213–226.

[40] Calzarossa, M. and D. Ferrari (1985) "A Sensitivity Study of the Clustering Approach to Workload Modeling (Extended Abstract)," *SIGMETRICS Perform. Eval. Rev.*, **13**(2), pp. 38–39.

[41] Hellerstein, J. L., F. Zhang, and P. Shahabuddin (2001) "A Statistical Approach to Predictive Detection," *Comput. Networks*, **35**(1), pp. 77–95.

[42] Ferrari, D. (1972) "Workload Characterization and Selection in Computer Performance Measurement," *IEEE Computer*, **5**, pp. 18–24.

[43] Feitelson, D. G. (2002) "Workload Modeling for Performance Evaluation," in *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, Springer-Verlag, pp. 114–141.

[44] Diao, Y., J. L. Hellerstein, S. Parekh, H. Shaikh, and M. Surendra (2006) "Controlling Quality of Service in Multi-Tier Web Applications," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, IEEE Computer Society, pp. 25–25.

[45] Riska, A. and E. Riedel (2006) "Disk Drive Level Workload Characterization," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, USENIX Association, pp. 9–9.

[46] Chen, Y., A. S. Ganapathi, R. Griffith, and R. H. Katz (2010) *Analysis and Lessons from a Publicly Available Google Cluster Trace, Tech. Rep. UCB/EECS-2010-95*, UC Berkeley.

[47] Joseph Hellerstein, "Google Cluster Data," `http://tinyurl.com/google-cluster-data`.

[48] Kavulya, S., J. Tan, R. Gandhi, and P. Narasimhan (2010) "An Analysis of Traces from a Production MapReduce Cluster," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, IEEE Computer Society, pp. 94–103.

[49] Chen, Y., A. S. Ganapathi, R. Griffith, and R. H. Katz (2010) *A Methodology for Understanding MapReduce Performance Under Diverse Workloads, Tech. Rep. UCB/EECS-2010-135*, UC Berkeley.

[50] Chen, Y., S. Alspaugh, and R. Katz (2012) "Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads," *Proc. VLDB Endow.*, **5**(12), pp. 1802–1813.

[51] Kao, W.-l. and R. K. Iyer (1992) "A User-Oriented Synthetic Workload Generator," in *Proceedings of the International Conference on Distributed Computing Systems, ICDCS*, pp. 270–277.

[52] Galindo, H. E. S., W. M. Santos, P. R. M. Maciel, B. Silva, S. M. L. Galdino, and J. P. Pires (2009) "Synthetic Workload Generation for Capacity Planning of Virtual Server Environments," in *Proceedings of the 2009 IEEE international conference on Systems, Man and Cybernetics*, SMC'09, IEEE Press, pp. 2837–2842.

[53] Jin, S. and A. Bestavros (2001) "GISMO: A Generator of Internet Streaming Media Objects and Workloads," *ACM SIGMETRICS Performance Evaluation Review*, **29**(3).

[54] Li, H., W.-C. Lee, A. Sivasubramaniam, and L. Giles (2007) "SearchGen: A Synthetic Workload Generator for Scientific Literature Digital Libraries and Search Engines," in *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '07, ACM, pp. 137–146.

[55] Chen, Y., A. Ganapathi, R. Griffith, and R. H. Katz (2010) *Towards Understanding Cloud Performance Tradeoffs Using Statistical Workload Analysis and Replay, Tech. Rep. UCB/EECS-2010-81*, UC Berkeley.

[56] Ganapathi, A., Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson (2009) *Statistics-Driven Workload Modeling for the Cloud*, *Tech. Rep. UCB/EECS-2009-160*, UC Berkeley.

[57] Sobel, W., S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson (2008) "Cloudstone: Multi-platform, Multi-language Benchmark and Measurement Tools for Web 2.0," in *Proceedings of Workshop on Cloud Computing*, CCA '08.

[58] Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears (2010) "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, ACM, pp. 143–154.

[59] Gridmix, `http://tinyurl.com/gridmixbenchmark`.

[60] Huang, S., J. Huang, J. Dai, T. Xie, and B. Huang (2010) "The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 41–51.

[61] Kim, K., K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom (2008) "MRBench: A Benchmark for MapReduce Framework," in *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, ICPADS '08, IEEE Computer Society, pp. 11–18.

[62] Raman, R., M. Livny, and M. Solomon (1998) "Matchmaking: Distributed Resource Management for High Throughput Computing," in *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, HPDC '98, IEEE Computer Society, pp. 140–146.

[63] Czajkowski, K., I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke (1998) "A Resource Management Architecture for Metacomputing Systems," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98, Springer-Verlag, pp. 62–82.

[64] Polo, J., D. Carrera, Y. Becerra, M. Steinder, and I. Whalley (2010) "Performance-Driven Task Co-Scheduling for MapReduce Environments," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium, NOMS*, IEEE, pp. 373–380.

[65] Qin, A., D. Tu, C. Shu, and C. Gao (2009) "XConveryer: Guarantee Hadoop Throughput via Lightweight OS-Level Virtualization,"

in *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, GCC '09, IEEE Computer Society, pp. 299–304.

[66] Tian, C., H. Zhou, Y. He, and L. Zha (2009) "A Dynamic MapReduce Scheduler for Heterogeneous Workloads," in *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, GCC '09, IEEE Computer Society, pp. 218–224.

[67] Verma, A., L. Cherkasova, and R. H. Campbell (2011) "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, ACM, pp. 235–244.

[68] "Capacity Scheduler," `http://tinyurl.com/scheduler-capacity`.

[69] "Fair Scheduler," `http://tinyurl.com/hadoopfairscheduler`.

[70] Zaharia, M., D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica *Job Scheduling for Multi-User MapReduce Clusters, Tech. Rep. UCB/EECS-2009-55*, UC Berkeley.

[71] Sandholm, T. and K. Lai (2010) "Dynamic Proportional Share Scheduling in Hadoop," in *Proceedings of the 15th international conference on Job scheduling strategies for parallel processing*, JSSPP'10, Springer-Verlag, pp. 110–131.

[72] Hindman, B., A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica (2011) "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, USENIX Association, pp. 22–22.

[73] Hadoop, "Next Generation MapReduce," `http://tinyurl.com/YarnHadoop`.

[74] Zaharia, M., A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica (2008) "Improving MapReduce Performance in Heterogeneous Environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, USENIX Association, pp. 29–42.

[75] Zaharia, M., D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica (2010) "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, ACM, pp. 265–278.

[76] Kc, K. and K. Anyanwu (2010) "Scheduling Hadoop Jobs to Meet Deadlines," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, IEEE Computer Society, pp. 388–392.

[77] "Hadoop on Vmware," `http://serengeti.cloudfoundry.com/`.

[78] Cardosa, M., P. Narang, A. Chandra, H. Pucha, and A. Singh (2011) "STEAMEngine: Driving MapReduce Provisioning in the Cloud," in *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, IEEE Computer Society, pp. 1–10.

[79] Chohan, N., C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz (2010) "See Spot Run: Using Spot Instances for Mapreduce Workflows," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, USENIX Association, pp. 7–7.

[80] Sandholm, T. and K. Lai (2009) "MapReduce Optimization using Regulated Dynamic Prioritization," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, ACM, pp. 299–310.

[81] Ibrahim, S., H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi (2009) "Evaluating MapReduce on Virtual Machines: The Hadoop Case," in *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, Springer-Verlag, pp. 519–528.

[82] Vmware, "Elastic Hadoop for Cloud," `http://tinyurl.com/elastic-hadoop`.

[83] Kang, H., Y. Chen, J. L. Wong, R. Sion, and J. Wu (2011) "Enhancement of Xen's Scheduler for MapReduce Workloads," in *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, ACM, pp. 251–262.

[84] Clay, R. B. (2011) *Enabling MapReduce to Harness Idle Cycles in Interactive-User Clouds*, Master's thesis, NC State University.

[85] Bu, X., C. Xu, and J. Rao (2012) *Interference and Locality-Aware Scheduling for MapReduce Service in Virtual Clusters*, *Tech. rep.*, Univ. of Colarado.

[86] Nathuji, R., A. Kansal, and A. Ghaffarkhah (2010) "Q-clouds: Managing Performance Interference Effects for QoS-Aware Clouds," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, ACM, pp. 237–250.

[87] Chiang, R. C. and H. H. Huang (2011) "TRACON: Interference-Aware Scheduling for Data-Intensive Applications in Virtualized Environments," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM, pp. 47:1–47:12.

[88] Koh, Y., R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu (2007) "An Analysis of Performance Interference Effects in Virtual Environments," in *Proceedings of the Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pp. 200 –209.

[89] Pu, X., L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, Y. Cao, and L. Liu (2012) "Who is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds," *Services Computing, IEEE Transactions on*, **PP**(99), p. 1.

[90] Lin, H., X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang (2010) "MOON: MapReduce On Opportunistic eNvironments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, ACM, pp. 95–106.

[91] Kooburat, T. and M. Swift (2011) "The Best of Both Worlds with On-Demand Virtualization," in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, USENIX Association, pp. 4–4.

[92] IBM, "Tivoli," `http://www.ibm.com/tivoli`.

[93] Corporation, H., "H.P. Openview," `http://www.openview.hp.com`.

[94] Ganglia, "Monitoring tool," `http://ganglia.info`.

[95] Nagios, `http://www.nagios.org`.

[96] Wang, C., V. Talwar, K. Schwan, and P. Ranganathan (2010) "Online Detection of Utility Cloud Anomalies using Metric Distributions," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium, NOMS 2010, 19-23 April 2010, Osaka, Japan*, IEEE, pp. 96–103.

[97] Agarwala, S., F. Alegre, K. Schwan, and J. Mehalingham (2007) "E2EProf: Automated End-to-End Performance Management for Enterprise Systems," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, IEEE Computer Society, pp. 749–758.

[98] Chen, M. Y., E. Kiciman, E. Fratkin, A. Fox, and E. Brewer (2002) “Pinpoint: Problem Determination in Large, Dynamic Internet Services,” in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN ’02, IEEE Computer Society, pp. 595–604.

[99] Kang, S., Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song (2010) “Orchestrator: An Active Resource Orchestration Framework for Mobile Context Monitoring in Sensor-Rich Mobile Environments,” in *PerCom*, pp. 135–144.

[100] Pelleg, D., M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan (2008) “Vigilant: Out-of-Band Detection of Failures in Virtual Machines,” *SIGOPS Oper. Syst. Rev.*, **42**(1), pp. 26–31.

[101] Ben-Yehuda, M., D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg (2009) “NAP: A Building Block for Remediating Performance Bottlenecks via Black Box Network Analysis,” in *Proceedings of the 6th international conference on Autonomic computing*, ICAC ’09, ACM, pp. 179–188.

[102] Kc, K. and X. Gu (2011) “ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures,” in *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS ’11, IEEE Computer Society, pp. 11–20.

[103] Cherkasova, L., K. M. Ozonat, N. Mi, J. Symons, and E. Smirni (2008) “Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change.” in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, IEEE Computer Society, pp. 452–461.

[104] Bare, K. A., S. Kavulya, and P. Narasimhan (2010) “Hardware Performance Counter-Based Problem Diagnosis for E-Commerce Systems.” in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium, NOMS*, IEEE, pp. 551–558.

[105] Kang, H., X. Zhu, and J. L. Wong (2012) “DAPA: Diagnosing Application Performance Anomalies for Virtualized Infrastructures,” in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE’12, USENIX Association, pp. 8–8.

[106] Tan, Y., H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan (2012) “PREPARE: Predictive Performance Anomaly Prevention

for Virtualized Cloud Systems," in *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems*, ICDCS '12, IEEE Computer Society, pp. 285–294.

[107] Kleinrock, L. (1975) *Queueing Systems*, 2nd ed., Wiley-Interscience.

[108] Hartigan, J. A. and M. A. Wong (1979) "A K-Means Clustering Algorithm," *Applied Statistics*.

[109] Sharma, B., V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das (2011) *Modeling and synthesizing task placement constraints in Google compute clusters*, *Tech. Rep. CSE#11-005*, Computer Science and Engineering Department, Pennsylvania State University.

[110] Denneulin, Y., E. Romagnoli, and D. Trystram (2004) "A Synthetic Workload Generator for Cluster Computing." in *IPDPS*, IEEE Computer Society.

[111] "Powered-by-Hadoop," `http://wiki.apache.org/hadoop/PoweredBy`.

[112] "Hadoop Code," `http://svn.apache.org/viewvc/hadoop/common/trunk`.

[113] "Hadoop YARN," `http://tinyurl.com/hadoop-yarn`.

[114] Apache, "Hadoop Profiler: Collecting CPU and Memory Usage for MapReduce Tasks," `https://issues.apache.org/jira/browse/MAPREDUCE-220`.

[115] lxc, "Linux Control Groups," `http://en.wikipedia.org/wiki/Cgroups`.

[116] "Xen Hypervisor," `http://www.xen.org`.

[117] Gridmix2, `http://tinyurl.com/gridmix-v2`.

[118] "Wikitrends," `http://trendingtopics.org`.

[119] "Twitter traces," `http://an.kaist.ac.kr/traces/WWW2010.html`.

[120] Menon, A., J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel (2005) "Diagnosing Performance Overheads in the Xen Virtual Machine Environment," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, ACM, pp. 13–23.

[121] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003) "Xen and the Art of Virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, ACM, pp. 164–177.

[122] "No Virtualization for Clouds," `http://tinyurl.com/novirtualization`.

[123] Herodotou, H. and S. Babu (2011) "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs." *PVLDB*, **4**(11), pp. 1111–1122.

[124] Sharma, B., R. Prabhakar, S. Lim, M. Kandemir, and C. Das (2012) "MROrchestrator: A Fine-Grained Resource Orchestration Framework for MapReduce Clusters," in *Proceedings of the IEEE 5th International Conference on Cloud Computing (CLOUD)*, pp. 1 –8.

[125] "Cgroup support for I/O throttling," `http://tinyurl.com/io-throttle`.

[126] Hadoop, "Fair Scheduler," `http://tinyurl.com/scheduler-fair`.

[127] RUBiS, "E-commerce Application," `http://rubis.ow2.org`.

[128] TPC-W, "E-commerce Benchmarking." `http://www.tpc.org/tpcw`.

[129] Olio, `http://radlab.cs.berkeley.edu/wiki/Olio_Deployment`.

[130] Kutare, M., G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf (2010) "Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers," in *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, ACM, pp. 141–150.

[131] Bodik, P., M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen (2010) "Fingerprinting the Datacenter: Automated Classification of Performance Crises," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, ACM, pp. 111–124.

[132] Wood, T., E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani (2010) "Disaster Recovery as a Cloud Service: Economic Benefits & Deployment Challenges," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, USENIX Association, pp. 8–8.

[133] "Amazon Outage Cloud Disaster," `tinyurl.com/amazon-outage`.

[134] Cherkasova, L., K. Ozonat, N. Mi, J. Symons, and E. Smirni (2009) "Automated Anomaly Detection and Performance Modeling of Enterprise Applications," *ACM Trans. Comput. Syst.*, **27**(3), pp. 6:1–6:32.

[135] Nguyen, H., Y. Tan, and X. Gu (2011) "PAL: Propagation-aware Anomaly Localization for Cloud Hosted Distributed Applications," in *Proceedings of the Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, ACM, pp. 1–8.

[136] Wang, C., K. Viswanathan, C. Lakshminarayan, V. Talwar, W. Satterfield, and K. Schwan (2011) "Statistical Techniques for Online Anomaly Detection in Data Centers," in *Proceedings of the 12th International Conference on Integrated Network Management*, pp. 385–392.

[137] Verma, A., P. Ahuja, and A. Neogi (2008) "pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, Springer-Verlag, pp. 243–264.

[138] Verma, A., G. Kumar, R. Koller, and A. Sen (2011) "CosMig: Modeling the Impact of Reconfiguration in a Cloud," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, IEEE Computer Society, pp. 3–11.

[139] Gao, J., G. Jiang, H. Chen, and J. Han (2009) "Modeling Probabilistic Measurement Correlations for Problem Determination in Large-Scale Distributed Systems," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, IEEE Computer Society, pp. 623–630.

[140] "KNN Classifier," `http://tinyurl.com/knn-wiki`.

[141] Zhang, S., I. Cohen, J. Symons, and A. Fox (2005) "Ensembles of Models for Automated Diagnosis of System Performance Problems," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, IEEE Computer Society, pp. 644–653.

[142] Li, Z., X. Wang, Z. Liang, and M. K. Reiter (2008) "AGIS: Towards Automatic Generation of Infection Signatures," in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, IEEE Computer Society, pp. 237–246.

[143] VMware, "Vmkperf for VMware ESX 5.0, 2011," .

[144] "VMware Powercli Cmdlets," `http://tinyurl.com/powercli-cmd`.

[145] Chandola, V., A. Banerjee, and V. Kumar (2009) "Anomaly Detection: A Survey," *ACM Comput. Surv.*, **41**(3), pp. 15:1–15:58.

[146] Sharma, B., P. Jayachandran, A. Verma, and C. Das (2012) HybridMR*: A Framework for Problem Determination and Diagnosis in Shared Dynamic Clouds*, *Tech. Rep. CSE #12-001*, Penn State.

[147] Tumanov, A., J. Cipar, G. R. Ganger, and M. A. Kozuch (2012) "alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, ACM, pp. 25:1–25:7.

[148] Sharma, B., V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das (2011) "Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, ACM, pp. 3:1–3:14.

[149] "IBM Smart Cloud," `http://www.ibm.com/cloud-computing`.

Vita

Bikash Sharma

Bikash Sharma is a Ph.D candidate in the department of Computer Science and Engineering at the Pennsylvania State University. Bikash finished his Bachelor of Technology (with Honors) in Computer Science and Engineering from National Institute of Technology, Rourkela, India in summer 2007, and right after joined Penn State in Fall 2007, for his doctoral studies. His research interests broadly include distributed systems, cloud computing, Internet data centers, virtualization, big data analytics, networks, and Web 2.0 technologies. He has published in revered conferences including IEEE DSN, ACM SOCC, IEEE ICDCS, IEEE ISPASS, IEEE CLOUD and IEEE CLUSTER. Bikash has worked as a research intern in IBM Research in summer 2011, and as software engineer intern at Google Inc. during summers of 2010 and 2012. Bikash enjoys playing various kinds of racket sports, participating in programming challenges, traveling, and learning new technologies.