8-2016

Towards An Efficient Cloud Computing System: Data Management, Resource Allocation and Job Scheduling

Jinwei Liu
Clemson University, jinweil@g.clemson.edu

Towards An Efficient Cloud Computing System: Data Management, Resource Allocation and Job Scheduling

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Jinwei Liu
August 2016

Accepted by:
Professor Haiying Shen, Committee Chair
Professor Kuang-Ching Wang
Professor Adam Hoover
Professor James (Zijun) Wang

Abstract

Cloud computing is an emerging technology in distributed computing, and it has proved to be an effective infrastructure to provide services to users. Cloud is developing day by day and faces many challenges. One of challenges is to build cost-effective data management system that can ensure high data availability while maintaining consistency. Another challenge in cloud is efficient resource allocation which ensures high resource utilization and high SLO availability. Scheduling, referring to a set of policies to control the order of the work to be performed by a computer system, for high throughput is another challenge. In this dissertation, we study how to manage data and improve data availability while reducing cost (i.e., consistency maintenance cost and storage cost); how to efficiently manage the resource for processing jobs and increase the resource utilization with high SLO availability; how to design an efficient scheduling algorithm which provides high throughput, low overhead while satisfying the demands on completion time of jobs.

Replication is a common approach to enhance data availability in cloud storage systems. Previously proposed replication schemes cannot effectively handle both correlated and non-correlated machine failures while increasing the data availability with the limited resource. The schemes for correlated machine failures must create a constant number of replicas for each data object, which neglects diverse data popularities and cannot utilize the resource to maximize the expected data availability. Also, the previous schemes neglect the consistency maintenance cost and the storage cost caused by replication. It is critical for cloud providers to maximize data availability hence minimize SLA (Service Level Agreement) violations while minimize cost caused by replication in order to maximize the revenue. In this dissertation, we build a nonlinear programming model to maximize data availability in both types of failures and minimize the cost caused by replication. Based on the model's solution for the replication degree of each data object, we propose a low-cost multi-failure resilient replication scheme (MRR). MRR can effectively handle both correlated and

non-correlated machine failures, considers data popularities to enhance data availability, and also tries to minimize consistency maintenance and storage cost.

In current cloud, providers still need to reserve resources to allow users to scale on demand. The capacity offered by cloud offerings is in the form of pre-defined virtual machine (VM) configurations. This incurs resource wastage and results in low resource utilization when the users actually consume much less resource than the VM capacity. Existing works either reallocate the unused resources with no Service Level Objectives (SLOs) for availability[1] or consider SLOs to reallocate the unused resources for long-running service jobs. This approach increases the allocated resource whenever it detects that SLO is violated in order to achieve SLO in the long term, neglecting the frequent fluctuations of jobs' resource requirements in real-time application especially for short-term jobs that require fast responses and decision making for resource allocation. Thus, this approach cannot fully utilize the resources to process data because they cannot quickly adjust the resource allocation strategy dealing with the fluctuations of jobs' resource requirements. What's more, the previous opportunistic based resource allocation approach aims at providing long-term availability SLOs with good QoS for long-running jobs, which ensures that the jobs can be finished within weeks or months by providing slighted degraded resources with moderate availability guarantees, but it ignores deadline constraints in defining Quality of Service (QoS) for short-lived jobs requiring online responses in real-time application, thus it cannot truly guarantee the QoS and long-term availability SLOs. To overcome the drawbacks of previous works, we adequately consider the fluctuations of unused resource caused by bursts of jobs' resource demands, and present a cooperative opportunistic resource provisioning (CORP) scheme to dynamically allocate the resource to jobs. CORP leverages complementarity of jobs' requirements on different resource types and utilizes the job packing to reduce the resource wastage and increase the resource utilization.

An increasing number of large-scale data analytics frameworks move towards larger degrees of parallelism aiming at high throughput. Scheduling that assigns tasks to workers and preemption that suspends low-priority tasks and runs high-priority tasks are two important functions in such frameworks. There are many existing works on scheduling and preemption in literature to provide high throughput. However, previous works do not substantially consider dependency in increasing throughput in scheduling or preemption. Considering dependency is crucial to increase the overall

[1] Availability refers to the probability of an allocated resource being remain operational and accessible during the validity of the contract [34].

throughput. Besides, extensive task evictions for preemption increase context switches, which may decrease the throughput. To address the above problems, we propose an efficient scheduling system Dependency-aware Scheduling and Preemption (DSP) to achieve high throughput in scheduling and preemption. First, we build a mathematical model to minimize the makespan with the consideration of task dependency, and derive the target workers for tasks which can minimize the makespan; second, we utilize task dependency information to determine tasks' priorities for preemption; finally, we present a probabilistic based preemption to reduce the numerous preemptions, while satisfying the demands on completion time of jobs.

We conduct trace driven simulations on a real-cluster and real-world experiments on Amazon S3/EC2 to demonstrate the efficiency and effectiveness of our proposed system in comparison with other systems. The experimental results show the superior performance of our proposed system. In the future, we will further consider data update frequency to reduce consistency maintenance cost, and we will consider the effects of node joining and node leaving. Also we will consider energy consumption of machines and design an optimal replication scheme to improve data availability while saving power. For resource allocation, we will consider using the greedy approach for deep learning to reduce the computation overhead caused by the deep neural network. Also, we will additionally consider the heterogeneity of jobs (i.e., short jobs and long jobs), and use a hybrid resource allocation strategy to provide SLO availability customization for different job types while increasing the resource utilization. For scheduling, we will aim to handle scheduling tasks with partial dependency, worker failures in scheduling and make our DSP fully distributed to increase its scalability. Finally, we plan to use different workloads and real-world experiment to fully test the performance of our methods and make our preliminary system design more mature.

Acknowledgments

I would like to express my sincere appreciation and thanks to my advisor Dr. Haiying Shen, whose passion and earnest manner in research have transformed me and will benefit me for the rest of my life. Her guidance helped me overcome numerous difficulties and finish my Ph.D. study.

Dr. Haiying Shen has an extraordinarily lucid thought process. She is incredibly creative and her depth of understanding of computer systems and technology is truly astonishing. Dr. Haiying Shen is really a hard-working professor, and she inspired me by being a role model herself. I greatly enjoyed her advice and insights on my personal and professional life. Her effort and patience will never be forgotten.

I would also like to thank my committee members: Dr. Kuang-Ching Wang, Dr. Adam Hoover, and Dr. James (Zijun) Wang, not only for their time and extreme patience, but also for their valuable advice, comments, and immeasurable assistance.

I thank my colleagues in the Pervasive Communications Laboratory: Ze Li, Kang Chen, Guoxin Liu, Chenxi Qiu, Bo Wu, Yuhua, Lin, Liuhua Chen, Zhuozhao Li, Li Yan, Ankur Sarker, Haoyu Wang, Mehdi Rahmani-andebili, for helping me on my research and also daily life. They are wonderful companies, and together we went through all good times and hard times.

I especially thank my beloved parents for their constant support and unconditional love. They have taught me about hard work, self-respect and persistence, about how to be independent. They always encouraging me to continue my studies and follow a career path I loved. I will forever be thankful to all my families who are the sources of all my happiness.

Many other people have provided support and feedback for my work, including Dr. Warren Adams, Dr. Svetlana Poznanovikj, Dr. Shuhong Gao, Matthew Saltzman, Dr. Harlan Russell, Dr. Jason O. Hallstrom, Dr. Hongxin Hu, Dr. Brain C. Dean, Dr. Amy Apon, Dr. Harold C. Grossman and D. E. Stevenson. I would like to thank them. Finally, I want to thank Dr. John Wilkes who

provides the Google trace which supports the experiment of the work. I also wan to thank the reviewers from SOCC 2014, NSDI 2015, ATC 2015, SC 2015, INFOCOM 2016 and CLUSTER 2016 for their helpful feedback.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As an emerging technology in distributed computing, cloud computing has proved to be an effective infrastructure to provide services to users. Cloud is comprised of a collection of virtual machines including both computational and storage facility. Cloud is developing day by day and faces many challenges. One of challenges is to build cost-effective data management system ensuring high data availability while maintaining consistency [15]. Another challenge in cloud is efficient resource allocation which ensures high resource utilization [123]. Scheduling jobs (especially parallel jobs) with constraints for high throughput is another challenge [14] in cloud. To address these challenges, in this dissertation, we aim to study how to manage data and improve data availability while reducing cost (i.e., consistency maintenance cost and storage cost); how to efficiently manage the resource for processing jobs and increase the resource utilization; how to design an efficient scheduling algorithm which provides high throughput, low overhead while satisfying the demands on completion time of jobs. Below we briefly introduce how the three problems come out.

Datacenter storage systems (e.g., Hadoop Distributed File System (HDFS) [111], RAMCloud [87], Google File System (GFS) [55] and Windows Azure [33]) are an important component of cloud datacenters, especially for data-intensive services in this big data era. It is critical for cloud providers to reduce the violations of Service Level Agreements (SLAs) for tenants to provide high quality-of-service and avoid the associated penalties. For example, a typical SLA required from services that use Amazon Dynamo storage system is that 99.9% of the read and write requests execute within 300ms [46]. Therefore, a storage system must guarantee data availability for different applications to comply to SLAs, which however is a non-trivial task. In many cloud services such

as Amazon, Google App Engine and Windows Azure, the server failure probability is in the range of [1%, 10%] and the corresponding data availability is in the range of [99.9%, 99.99%] [99].

Data availability is usually influenced by data loss, which is typically caused by machine failures including correlated and non-correlated machine failures coexisting in storage systems. The former means multiple nodes fail (nearly) simultaneously, while the latter means nodes fail individually. Correlated machine failures often occur in large-scale storage systems [58, 86, 139] due to common failure causes (e.g., cluster power outages, workload-triggered software bug manifestations, Denial-of-Service attacks). For example, in cluster power outages [35, 44, 53], a non-negligible percentage (0.5%-1%) of nodes [35, 111] do not come back to life after power is restored [42]. Correlated machine failures cause significant data loss [42], which have been documented by Yahoo! [111], LinkedIn [35] and Facebook [30]. Non-correlated machine failures [139] are caused by reasons such as different hardware/software compositions and configurations, and varying network access ability. Measurements of over 10,000 file systems on desktop computers at Microsoft [27] demonstrate machines experience disk head crashes hence permanent data-loss failures in a temporally uncorrelated fashion. Non-correlated failures can be classified into uniform and nonuniform machine failures, in which machines fail with the same and different probabilities (possibly due to the same or different computer configurations), respectively.

Many cloud providers offer resources for leasing with elastic under infrastructure as a service (IaaS) paradigm. Although the elastic and on-demand nature of cloud computing can help cloud users meet their dynamic and fluctuating demands with minimal management overhead, the users are still allocated the resource more than their demands on the amount of resources for processing their jobs, resulting in resource wastage [51, 109]. Also, the elasticity proposed by cloud computing does not really come for free. Providers have to reserve resources to allow users to scale on demand, and cope with workload variations. In practical scenarios, at any point in time, cloud providers impose a resource ceiling for each user to bound the maximum amount of resources that the user will be granted. Users usually get a small ceiling by default [34]. It is wasteful to provision enough capacity to guarantee that all users can be granted their ceilings all the time [34]. This is because for most applications, the average resource requirement is much lower than the peak [100]. Figure 1.1 shows an example of temporally unused resources in which the resources, such as CPU and memory (MEM in short), are idle (unused) and thus are wasted. Normal cloud approaches on resource allocation cannot be directly applied to dealing with real-time job processing for achieving high

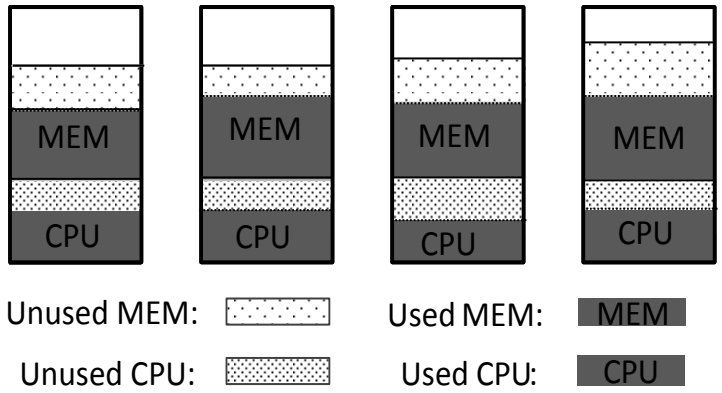


Figure 1.1: Resource wastes in terms of different types of temporarily unused resource.

resource utilization and low SLO violation rate because they either allocate the resource based on reservation or demand-based resource allocation and cannot fully utilize the resource all the time [34] and cannot well meet the requirements of real-time application (e.g., time-varying user demand on resource).

An increasing number of large scale analytic frameworks move towards high degree of parallelism aiming at providing high throughput. For example, MapReduce is a framework designed to process a large amount of data in the parallel manner on a cluster of computing nodes. In such a framework, each job is partitioned to tasks and run on the cluster servers in parallel. Task scheduling and preemption are two important functions for high job performance [73, 90, 130]. Job scheduling is the process of assigning jobs to worker machines in a manner to optimize the job performance. Usually, a user or a system submits jobs to the scheduler, which divides each job into tasks and forwards tasks to workers for processing. A job usually consists of hundreds or thousands of concurrent tasks [89]. Job completion time is determined by the completion time of the tail task. Placing a task on a contended machine extends the completion time of the task. Therefore, the tasks of each job should be scheduled to appropriate workers so that the job completion time can be reduced. Each worker has a waiting queue which is used for queuing tasks when a worker is allocated to more tasks than it can run concurrently. In a preemption method, the priorities of tasks are determined, and each worker chooses a high-priority task in its waiting queue to preempt its low-priority running task. The priorities are determined based on factors such as task's remaining time and waiting time to serve different performance objectives (e.g., high throughput, short job completion times).

Dependency usually exists among tasks of a job. That means, a task cannot start running

until its precedent tasks complete.[1] Many previous works have been proposed to detect the dependency between tasks [57, 82, 118, 121, 138]. However, previous scheduling algorithms and preemption algorithms do not coherently consider dependency and utilize the dependency information to increase throughput in scheduling or preemption by prioritizing tasks that will subsequently enable the execution of more dependent tasks. Previous scheduling algorithms [4, 16, 17, 19, 20, 23, 40, 90, 91, 96, 120, 122, 125, 126, 135] to increase throughput can be roughly classified into two categories. The works [16, 91, 122, 126, 135] focus on increasing the throughput by utilizing the data locality, that is, placing tasks on or close to machines that store the input data to reduce transmission overhead. However, the above works do not consider task dependency, which is a main factor to consider in scheduling for high throughput. Assigning tasks with the dependency relation at the same time leads to resource wastes since dependent tasks cannot run until their precedent tasks complete. The work [57] aims to maximize the throughput by fully utilizing the server resources. It packs tasks to machines based on the alignment score (the weighted dot product value between the vector of a machine's available resources and the task's peak usage of resources). Though this paper mentions task dependency, it does not provide a method to substantially consider dependency in scheduling. To simply consider dependency in scheduling, the scheduler can first schedule runnable tasks, and leave the dependent tasks to the next scheduling. However, the precedent tasks may complete a certain time period before the next scheduling, then the server resources during this time period cannot be fully utilized. Also, if at next scheduling time t_1, there will be three idle slots, then assigning a task with three dependent tasks rather than a task with zero or four dependent tasks at scheduling time t_0 can more fully utilize resources. Only considering currently runnable tasks in scheduling without considering the dependent tasks in a global view may not be able to fully increase the throughput. Thus, it cannot fully utilize task dependency information to increase the throughput by judiciously determining tasks' execution order so that more tasks have chance to start executing after a selected task finishes execution. Figure 1.2 shows the diverse dependency relations among tasks. Tasks T_2-T_5 cannot start executing until task T_1 finishes its execution. Similarly, this applies to tasks T_6 and T_{15}: tasks T_7-T_{14} and T_{16}-T_{19} cannot start executing until T_6 and T_{15} finish their executions, respectively. In Figure 1.2, task T_6 has more dependent tasks than tasks T_1 and

[1] In this paper, we do not consider job dependency (i.e., cross-job dependency) [98] and focus on task dependency in which a dependent task cannot start running until its precedent tasks complete, though partial dependency exists in some scenarios. Also, we do not consider the scenario that tasks dynamically add new tasks and extend the task-dependency graph because it is not a common occurrence in cloud computing.

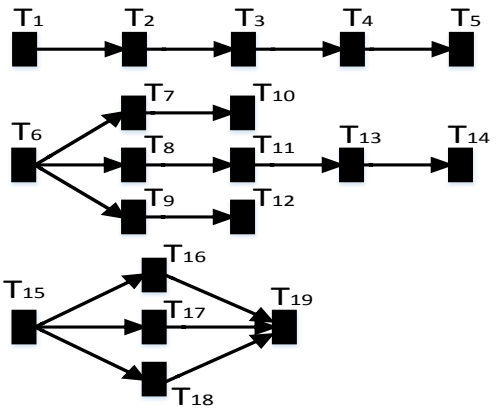


Figure 1.2: Diverse dependency relations among tasks.

T_{15}[2]. Execuitng T_6 at first can increase the chance that more dependent tasks can start executing after the precedent task T_6 finish its execution, which helps increase the throughput.

1.1 Problem Statement

1.1.1 Efficient Data Management for High Data Availability

Replication is a common approach to reduce data loss and enhance data availability. The consideration of data popularity in replication is also important to maximize the expected data availability[3]. Due to highly skewed data popularity distributions [18], popular data with considerably higher request frequency generates heavier load on the nodes, which may lead to data unavailability at a time. On the other hand, unpopular data with few requests wastes the resources for storing and maintaining replicas. Thus, an effective replication scheme must consider the diverse data popularities to use the limited resources (e.g., memory) [65] to increase expected data availability.

Though creating more replicas for a data object leads to higher data availability, it comes with higher consistency maintenance cost [28, 133] and storage cost [114]. In consistency maintenance, when a data is updated, an update is sent to its replica nodes. In addition to the number of replicas, the consistency maintenance cost and storage cost are also affected by the geographic distance and storage mediums (e.g., disk, SSD, EBS), respectively. To reduce the cost, we need to minimize the number of replicas, limit the geographic distance of replica nodes and replicate the

[2]For simplicity, the subscript of the symbol T only indicates the task number, which is different from the meaning explained in Table 4.1. We assume the task dependencies are known a priori [12, 52], though sometimes dependencies cannot be easily discovered ahead of time.

[3]We use a service-centric availability metric: the proportion of all successful service requests over all requests [139].

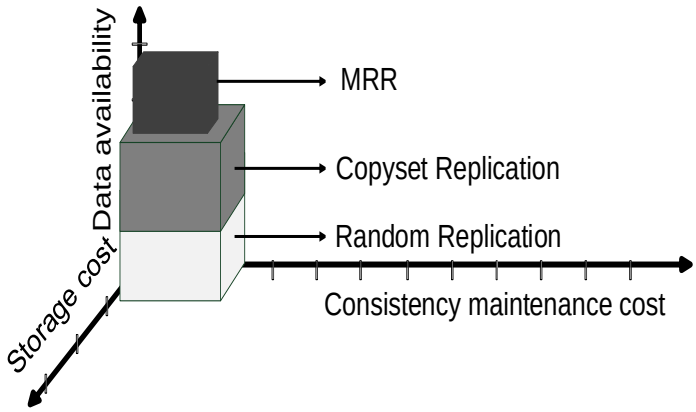


Figure 1.3: Achieving an optimal tradeoff among data availability, storage cost and consistency maintenance cost.

additional replicas beyond the required storage of applications to less expensive storage mediums while still provide SLA guarantee. Therefore, effective replication methods must maximize expected data availability (by considering both correlated and non-correlated machine failures and data popularity) and minimize the cost caused by replication (i.e., consistency maintenance cost and storage cost) [81, 133].

Random replication, as a popular replication scheme, has been widely used in datacenter storage systems including HDFS, RAMCloud, GFS and Windows Azure. These systems partition each data object into chunks (i.e., partitions), each of which is replicated to a constant number of randomly selected nodes on different racks. Though random replication can handle non-correlated machine failures, it cannot handle correlated machine failures well because data loss occurs if any combination of a certain number of nodes experience failures simultaneously [42]. Previously proposed data replication methods cannot effectively handle both correlated and non-correlated machine failures and utilize the limited resource to increase the data availability simultaneously [42, 53, 79, 86, 139]. Although many methods have been proposed to improve data availability [13, 28, 53, 67, 86, 139], they do not concurrently consider the data popularities and the cost caused by replication to increase expected data availability and decrease cost.

As shown in Figure 1.3, a key problem here is how to achieve an optimal tradeoff between increasing data availability and reducing cost caused by replication with the ultimate goal of SLA compliance and revenue maximization for cloud service providers.

1.1.2 Efficient Multi-Resource Allocation for High Resource Utilization

Many previous works [103, 109] try to increase resource utilization by predicting users' demand on the resource for their jobs. However their approaches are either reservation-based or demand-based resource allocation, which cannot fully utilize the resource all the time. The work [108] presents CloudScale to automate fine-grained elastic resource scaling for multi-tenant cloud computing infrastructures. CloudScale uses online resource demand prediction and prediction error handling to achieve adaptive resource allocation. Specifically, the model uses a fast Fourier transform (FFT) to identify repeating patterns (called signatures) to estimate future resource demands (if no repeating patterns is found, it adopts a discrete-time Markov chain to predict the resource demand in the near future), then the prediction error correction module performs online adaptive padding that adds a dynamically determined cushion value to the predicted resource demand to avoid under-estimation errors. However, the method in the work [108] is demand-based resource allocation which cannot fully utilize the resource because users typically get a small ceiling of resource from the resource providers. Another work [43] proposes a reservation-based scheduling consisting of two main processes: reservation and planning. Reservation process determines a job's resource needs and temporal requirements, and translates the job's completion service level agreements (SLA) into a service level objective (SLO) over predictable resource allocations, which is done ahead of job's execution. Planning process constructs a temporal assignment of cluster resources to jobs such that each job's RDL (reservation definition language which provides a uniform and abstract representation of all the jobs's needs) expression is satisfied. However, resource reservation can lead to resource underutilization as users do not necessarily understand complex physical resource requirements of jobs [48]. The work [48] presents Quasar, a cluster management system, to increase resource utilization while providing consistently high application performance. First, Quasar lets users communicate the performance constraints of the application in terms of throughput and/or latency, depending on the application type, and it determines the amount of the available resources needed to meet performance constraints based on users' expression of performance constraints. Second, Quasar relies on classification techniques to quickly and accurately determine the impact of the amount of resources (scale-out and scale-up), type of resources, and interference on performance for each workload. Third, Quasar performs resource allocation and assignment jointly. The classification results are used to determine the right amount and specific set of resources assigned to

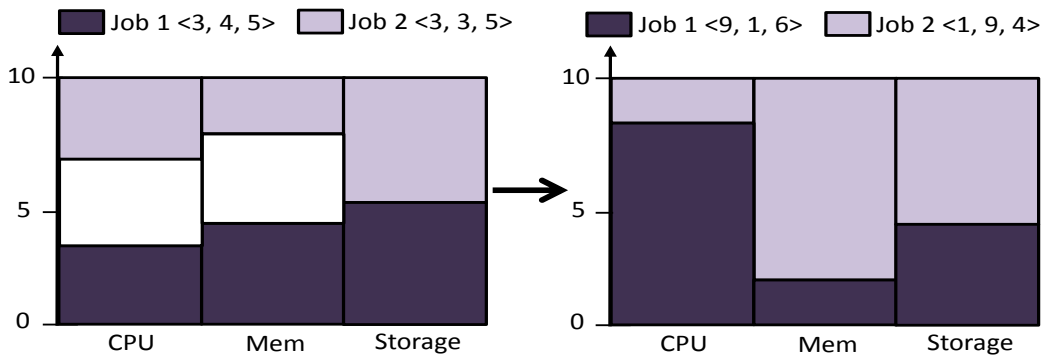


Figure 1.4: Allocate resource to jobs by leveraging complementarity of jobs' requirements on different resource types to increase resource utilization.

the workload. However, Quasar requires users to communicate performance constraints for each workload and it is still on-demand based resource allocation, and thus the resource cannot be fully utilized all the time because of the temporarily unused resource [34].

Most users can accept weaker promises. If all the users' loads do not peak at the same time, the cloud provider can benefit from statistical multiplexing to provision fewer resources than the sum of all users' ceilings [84], thereby fully utilizing the resource. A common approach of increasing resource utilization is to offer this temporarily-unused capacity in an opportunistic approach, in which the SLOs are not guaranteed [83]. The lack of SLOs restricts the applications that can benefit from these reclaimable resources. In order to achieve both high resource utilization and low SLO violation rate, some works [34, 83] and the product [3] present methods of offering resources opportunistically. The work [83] presents reusing unused cloud resources by offering leases in an opportunistic and preemptible way with no SLOs. Although the approach in the work [83] can increase the cloud utilization to some extent, they do not ensure SLOs, which affects the QoS because the resource can be preempted at any time. The product Amazon EC2 Spot Instances [3] offers opportunistic resources. The approach can increase the resource utilization, however, the offered resources have no SLOs, affecting QoS when the allocated resource for processing jobs is preempted. Although a recent work [34] presents the opportunistic approach to achieve long-term SLOs, the method cannot be applied to the application of processing short-lived jobs for achieving high resource utilization and low SLO violation rate. This is because i) it assumes that resource utilization pattern exists in training data of resource utilization of the jobs. Though this assumption may be true for long-lived jobs, it usually does not hold true for short-lived jobs (such as short-lived

queries in the applications of Internet-of-Things and online data processing that typically run for seconds or minutes [77, 78, 127]), in spite of the fact that short-lived jobs occupy most of the jobs in the cloud [47]; ii) it may result in resource fragmentation and thus lead to low resource utilization because it neglects jobs' resource intensity and may allocate much more resources to the jobs when it allocates multiple types of resources to users' jobs;[4] iii) it fails to adequately consider the fluctuations of unused resource caused by bursts of jobs' resource demands, and thus cannot well meet the requirement of time-varying user demand on resource when users' demand on resource for data processing changes frequently. In this paper, we aim to design a resource provisioning scheme for short-lived jobs in cloud with high resource utilization while achieving low SLO violation rate. The key challenges include: (1) how to accurately predict the amount of temporarily-unused resources of short-lived jobs with resource fluctuations? (2) how to more fully utilize the temporarily-unused resources by considering diverse resource intensities of jobs? (3) how to allocate the resource to short-live jobs to satisfy their time constraints.

1.1.3 Scheduling Parallel Jobs with constraints for High Throughput

Several existing works [19,22,40,59] focus on task preemption with the objective of maximizing throughput. The works in [19, 40, 59] choose the tasks that have the shortest remaining time to preempt, and the work in [22] further considers the waiting time to prevent task starvation, in which the task waits for a very long time due to the lower priority. However, none of the works considers utilizing the task dependency to increase the throughput[5]. Also, when the priority is too fine-grained in the priority space, there will be many preemptions. This leads to many context switchings and may reduce throughput. On the other hand, when the priority is too coarse-grained in the priority space, there will be long waiting time for the waiting tasks, leading to resource starvation. However, none of previous preemption works consider the tradeoff among the preemption overhead, throughput and the task waiting time while satisfy the demands on completion time of jobs. Therefore, scheduling jobs (especially parallel jobs) with constraints (e.g., dependency constraints) achieving high throughput and low overhead is still a problem [14] in cloud.

[4]The method in the work [34] does not consider how to leverage complementarity of jobs' requirements on different resource types because different intensive jobs (e.g., CPU intensive and Mem intensive) have different demands on different resource types, and it can easily result in resource fragmentation when we use the method in [34] to allocate multiple resource types to jobs with different resource intensities in terms of VMs.

[5]MapReduce Application Master implicitly considers dependency by first requesting resources for map tasks and then for reduce tasks, however it does not substantially utilize dependency to increase throughput in scheduling or preemption.

1.2 Research Approach

According to the discussion of the challenges in Section 1.1, the cloud providers need an efficient and cost-effective replication strategy to ensure high data availability, an efficient resource allocation strategy to allocate resource to jobs that can achieve high resource utilization, an efficient scheduling algorithm that can provide high throughput, low overhead while satisfying the demands on completion time of jobs. In this dissertation, we propose a low-cost multi-failure resilient replication scheme to ensure high data availability while reducing the cost (i.e., consistency maintenance cost and storage cost); we present a cooperative opportunistic resource provisioning strategy to dynamically allocate the resource to jobs for achieving high resource utilization; we present an efficient scheduling algorithm dependency-aware scheduling and preemption to achieve high throughput, low overhead while satisfying the demands on completion time of jobs. We briefly describe our solution for each problem below.

1.2.1 Low-Cost Multi-Failure Resilient Replication Scheme for High Data Availability in Cloud

To handle both correlated and non-correlated machine failures and improve data availability, we propose a low-cost multi-failure resilient replication scheme (MRR), which targets the application of distributed data-intensive services and storage system. MRR is more advantageous than previous replication schemes (e.g., Random Replication, Copyset Replication [42]) in that it can handle both correlated and non-correlated machine failures, and also jointly considers data popularity and the cost caused by replication. The steps of MRR are as follows:

(1) MRR builds a nonlinear programming model that aims to maximize expected data availability (in both types of failures) with the consideration of popularities and reduce the cost caused by replication. MRR creates more replicas for more popular data objects, and vice versa. To reduce storage cost, MRR stores data objects with lower priority (i.e., from lower priority applications and with larger sizes) to less expensive storage media, and vice versa. MRR then uses Lagrange multipliers to derive a solution: the replication degree (i.e., the number of replicas) of each data object.

(2) Based on the solution, MRR partitions all nodes to different groups with each group responsible

for replicating all data objects with the same replication degree. Then, MRR partitions nodes in a group into different sets [42]; each set consisting of nodes from different datacenters within a certain geographic distance. The replicas of a chunk are stored in the nodes in one set, so data loss occurs only if these nodes experience failures simultaneously. Each data chunk is replicated to the corresponding storage medium based on its priority. MRR reduces the frequency of data loss by reducing the number of sets in a group, i.e., the probability that all nodes in a set fail.

1.2.2 Cooperative Opportunistic Resource Management for High Resource Utilization

In order to provide a resource provisioning strategy for processing short-lived jobs with high resource utilization and low SLO violation rate, we present a deep learning based cooperative opportunistic resource provisioning (CORP) scheme to dynamically allocate the resource to jobs for achieving high resource utilization. The key steps of CORP are as follows:

(1) CORP uses the deep learning method to accurately predict the amount of temporarily-unused resource of each short-lived job, and it offers an opportunistic approach to reallocate predicted unused resources (with low SLO violation rate) in order to increase the resource utilization.

(2) CORP also considers the fluctuations of the amount of the unused resource caused by the peak and valley of jobs' resource demands. It first predicts the fluctuations of the amount of the unused resource using HMM, then adjusts the predicted amount for the peak and valley of the amount of the unused resource, and dynamically allocates the corrected amount of resource to jobs. CORP thus can adapt well to the requirement of time-varying user demand on resources.

(3) CORP uses a job packing strategy by leveraging complementary jobs' requirements on different resource types (e.g., CPU, MEM) and allocates such jobs to the same VM to fully utilize unused resource, which reduces the resource fragmentation and further increases the resource utilization (see Figure 1.4).

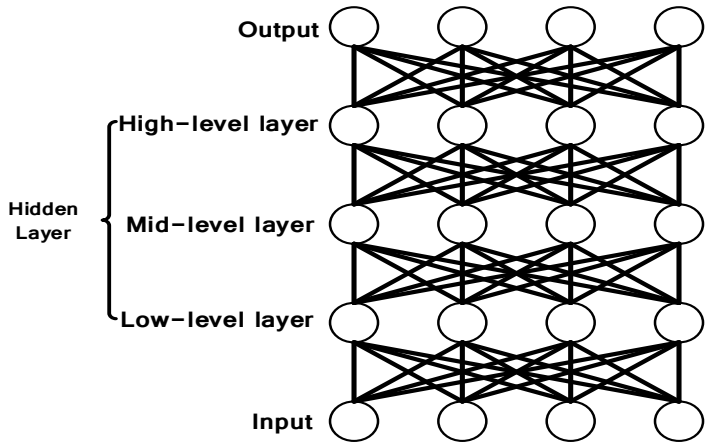


Figure 1.5: Deep neural networks.

1.2.3 Dependency-aware Scheduling and Preemption for High Throughput

To handle the problems in the previous scheduling and preemption methods, we propose Dependency-aware Scheduling and Preemption (DSP) to increase throughput and reduce overhead while satisfying the demands on complement jobs for scheduling jobs with constraints (e.g., dependency constraints, etc.). DSP includes the following key steps.

(1) DSP utilizes considers dependency and assigns tasks to workers so that independent tasks can run in parallel. To minimize the makespan, DSP builds a mathematical model, which can increase the throughput. Based on the model, the target workers and starting times for all tasks can be derived.

(2) DSP utilizes task dependency information to determine task priority. Specifically, DSP considers different dependency graphs, and assigns higher priority to tasks on which more tasks depend, ultimately, this approach increases the throughput by increasing the chance that more dependent tasks can start executing after the precedent tasks finish execution. Also, in preemption, DSP chooses a waiting task to preempt a running task considering the dependency between them to avoid dependency violation and increase throughput.

(3) Based on the derived task priority, DSP uses a probabilistic based preemption (PP) which selectively chooses waiting tasks with top priorities to preempt running tasks to reduce the time overhead (i.e., context switching) caused by preemption and hence increase the throughput.

1.3 Contributions

We summarize our expected contributions of the dissertation below:

- We build a nonlinear programming model that aims to maximize expected data availability (in both correlated and non-correlated machine failures) with the consideration of popularities and reduce the cost caused by replication. More replicas are created for more popular data objects, and vice versa. To reduce storage cost, data objects with lower priority (i.e., from lower priority applications and with larger sizes) are stored to less expensive storage mediums, and vice versa. We then use Lagrange multipliers to derive a solution: the replication degree (i.e., the number of replicas) of each data object. Based on the solution, we propose MRR to handle both correlated and non-correlated machine failures. MRR partitions all nodes to different groups with each group responsible for replicating all data objects with the same replication degree. Then, MRR partitions nodes in a group into different sets [42]; each set consisting of nodes from different datacenters within a certain geographic distance. The replicas of a chunk are stored in the nodes in one set, so data loss occurs only if these nodes experience failures simultaneously. Each data chunk is replicated to the corresponding storage medium based on its priority. MRR reduces the frequency of data loss by reducing the number of sets in a group, i.e., the probability that all nodes in a set fail.

- We propose a deep learning based cooperative opportunistic resource provisioning (CORP) scheme which offers the temporarily-unused resource in an opportunistic approach (i.e., The allocated unused resource can be preempted and reallocated to other new arriving jobs with a certain probability.) and increases the resource utilization. Moreover, CORP uses a resource packing strategy by leveraging complementarity of jobs' requirements on different resource types (e.g., CPU, MEM, etc.) and allocates resource to jobs (packing jobs to VMs), which reduces the resource fragmentation and further increase the resource utilization (see Figure 1.4). In addition, CORP adequately considers the fluctuations of unused resource caused by bursts of jobs' resource demands, it first predicts the fluctuations of unused resource using HMM, then it isolates peak and valley states of unused resource with error correction, and dynamically allocates the resource to jobs utilizing the unused resource. CORP thus can well meet the requirement of time-varying user demand on resources and increase the resource

utilization while reducing SLO violations.

- We propose a Dependency-aware Scheduling and Preemption (DSP), which consists of the following components: (i) DSP considers task dependency and assigns tasks to workers so that independent tasks can run in parallel. To minimize the makespan, a mathematical model is presented, which can increase the throughput. Based on the model, the target workers and starting times for all tasks can be derived; (ii) DSP utilizes task dependency information to determine task priority. Specifically, DSP considers different dependency graphs, and assigns higher priority to tasks on which more tasks depend, ultimately, this approach increases the throughput by increasing the chance that more dependent tasks can start executing after the precedent tasks finish execution. Also, DSP chooses a waiting task to preempt a running task considering the dependency between them to avoid dependency violation and increase throughput; (iii) Based on the derived task priority, DSP provides a probabilistic based preemption (PP) which selectively chooses waiting tasks with top priorities to preempt running tasks to reduce the time overhead (i.e., context switching) caused by preemption and hence increase the throughput.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 details the design of the low-cost multi-failure resilient replication scheme for high data availability in cloud storage. Chapter 3 describes the design of the CORP: cooperative opportunistic resource provisioning for short-lived jobs in cloud systems. Chapter 4 presents the design of the dependency-aware scheduling and preemption system. Chapter 5 reviews the related works. Finally, Chapter 6 concludes this dissertation with remarks on our future work.

Chapter 2

A Low-Cost Multi-Failure Resilient Replication Scheme for High Data Availability in Cloud Storage

In this chapter, we introduce our low-cost multi-failure resilient replication scheme (MRR) for high data availability in cloud storage. We first introduce the nonlinear integer programming (NLIP) model for MRR, which aims to maximize the expected data availability in both correlated and non-correlated machine failures. Then, we detail the design of our proposed MRR. Numerical results from trace parameters and experiments from real-world Amazon S3 show that MRR achieves high data availability, low data loss probability and low consistency maintenance cost and storage cost compared to previous replication schemes.

2.1 Nonlinear Integer Programming Model for MRR

A cloud storage system usually serves multiple applications simultaneously. Without loss of generality, we assume there are n applications in the cloud storage, and each application has m data objects [28]. Each data object belonging to an application is split into M partitions [28,131] and the data object is lost if any of its partitions is lost [42]. Replication degree of a data object represents the number of replicas of the data object. We use D_{ij} to denote the jth data object belonging to

application i (denoted by a_i). Let d_{ij} be the replication degree of D_{ij}. The replicas of a partition of a data object are placed in a set of d_{ij} different nodes. Suppose there are N servers in the cloud. For easy reference, Table 2.1 shows the main notations of MRR in this dissertation. For analytical tractability, we assume that a physical node (i.e., a server) belongs to a rack, a room, a datacenter, a country and a continent. To easily identify the geographic location of a physical node, each physical node has a label in the form of "continent-country-datacenter-room-rack-server" [28, 29].

Table 2.1: Notations in MRR

N	Total number of nodes	C_s^{th}	Upper bound of C_s
T	One epoch duration	φ_{ij}	Popularity function of D_{ij}
p	Probability of a server failure	ψ_{ij}	Priority function of D_{ij}
D_{ij}	The jth data in app. i	K_i	Space capacity for app. i
s_{ij}	The size of D_{ij}	r_{ij}	Probability of requesting D_{ij}
d_{ij}	Replication degree of D_{ij}	v_{ij}	Number of visits to D_{ij} / epoch
M	Number of chunks of each data	$C_{s_{ij}}$	D_{ij}'s storage space price
P_f	Probability of data loss	p_{uni}	Probability of data loss in
$\bar{P}_r$	Expected request failure		uniform machine failures
S	Scatter width	p_{cor}	Probability of data loss in
C_c	Consistency maint. cost		correlated machine failures
C_s	Total storage cost	p_{non}	Probability of data loss in non-
n	Number of data objects		uniform machine failures
C_c^{th}	Upper bound for C_c	P_r^{th}	Upper bound of $\bar{P}_r$

Problem Statement: Given data object request probabilities, data object sizes, space constraints for different applications, and thresholds for request failure probability, consistency maintenance cost and storage cost, what is the optimal replication degree for each data object, so that the request failure probability, consistency maintenance cost and storage cost are minimized in both correlated and non-correlated machine failures? Then, how to assign the chunk replicas of data objects to the nodes to achieve the objectives?

Our goal is to maximize the data availability in both correlated and non-correlated machine failures while reducing the costs (consistency maintenance cost and storage cost) caused by replication, and eventually build an effective and low-cost replication scheme for high data availability. In the following, Section 2.1.1 explains data availability maximization. We calculate the expected probability of data loss in correlated and non-correlated (uniform and nonuniform) machine failures, and finally calculate the expected probability of data loss in a cloud system with both types of failures. Section 2.1.2 explains the consistency maintenance cost minimization, and Section 2.1.3 explains the storage cost minimization. Section 2.1.4 formulates the problem by combining these

sub-problems and also gives an approach to obtain the solution.

2.1.1 Data Availability Maximization

We maximize data availability by considering both machine failure probability and data object request probability (i.e., popularity). We minimize the data loss probability in both correlated and non-correlated machine failures. Different data objects may have different popularities, and then have different demands on the number of replicas to ensure data availability.

2.1.1.1 Correlated Machine Failures

To reduce data loss in correlated machine failures, we adopt the fault-tolerant set (FTS) concept from [42], which is a distinct set of servers holding all copies of a given partition. Each FTS is a single unit of failure. That is, when an FTS fails, at least one data object is lost. For correlated machine failures, the probability of data loss increases as the number of FTSs increases because the probability that the failed servers constitute at least one FTS increases (It is more likely that the failed servers include at least one FTSs). So we minimize the probability of data loss by minimizing the number of FTSs.

To this end, we can statically assign each server to a single FTS, and constrain the replicas of a partition to a randomly selected preassigned FTS. That is, we first place the primary replica (i.e., original copy) on a randomly selected server, and then place the secondary replicas on all the nodes in this server's FTS. However, this will lead to load imbalance problem in which some servers become overloaded while some servers are underloaded due to data storage. On the other hand, random replication that randomly chooses a replica holder has a higher probability of data loss because almost every new replicated partition creates a distinct FTS. To achieve a tradeoff, we use the approach in Copyset Replication [42], which assigns a server to a limited number of FTSs rather than a single FTS. Due to the overlap of FTSs, after the primary replica is stored in a randomly selected server, the FTS options that can store the secondary replicas are those that hold the server. The servers in these sets are options that can be used to store the secondary replicas. The number of these servers is called *scatter width* (denoted by S). For example, if the FTSs that hold server 1 are {1,2,3}, {1,4,5}, then when the primary replica is stored at server 1, the secondary replica can be randomly placed either on servers 2 and 3 or 4 and 5. In this case the scatter width equals to 4.

The probability of correlated machine failures equals the ratio of the number of FTSs over

the maximum number of sets:

$$\frac{\#FTSs}{max\{\#sets\}} \tag{2.1}$$

To reduce the probability of data loss, Copyset Replication makes the replication scheme satisfy two conditions below:

- Condition 1: The FTSs overlap with each other by *at most* one server.
- Condition 2: The FTSs cover all the servers equally.

Copyset Replication uses the Balanced Incomplete Block Design (BIBD)-based method for any scatter width to create FTSs that satisfy both condition 1 and condition 2 and minimize the number of FTSs.[1]

We define a pair (X, A), where X is a set of servers in the system (i.e., $X = \{1, 2, ..., N\}$), and A is a collection of all FTSs in the system. Let N, R and λ be positive integers such that $N > R \geq 2$, BIBD for (N, R, λ) satisfies the following properties:

- Each FTS contains exactly R servers.
- Each pair of servers is contained in exactly λ FTSs.

When $\lambda = 1$, the BIBD provides an optimal design for minimizing the number of FTSs for scatter width $S = N - 1$. In this case, condition 1 ensures that each FTS increases the scatter width for its servers by exactly $R - 1$ compared to the case when $\lambda = 0$. Copyset Replication creates $\frac{S}{R-1}\frac{N}{R}$ FTSs. Then, the failure probability in correlated machine failures equals:

$$p_{cor} = \frac{\frac{S}{R-1}\frac{N}{R}}{\binom{N}{R}} \tag{2.2}$$

In random replication, the number of FTSs created is [42]:

$$\begin{cases} \#FTSs = N\binom{S}{R-1}, & S < \frac{N}{2} \\ \#FTSs \approx \binom{N}{R}, & S \approx N \end{cases} \tag{2.3}$$

Based on Equs. (2.1), (2.3), we can calculate the probability of correlated machine failures in random replication.

[1] In implementation, Copyset Replication uses random permutation to create FTSs.

Table 2.2: An example for BIBD-based method for $(12, 3, 1)$.

	0	1	2	3	4	5	6	7	8	9	10	11
B_1	1	1	1	0	0	0	0	0	0	0	0	0
B_2	0	0	0	1	1	0	0	0	0	0	1	0
B_3	0	0	0	0	0	0	1	1	1	0	0	0
B_4	0	0	0	0	0	0	0	0	0	1	1	1
B_5	1	0	0	1	0	0	0	0	1	0	0	0
B_6	0	1	0	0	1	0	0	1	0	0	0	0
B_7	0	0	1	0	0	1	0	0	0	0	0	1
B_8	0	0	0	0	0	1	1	0	0	1	0	0

We give an example to illustrate the process of generating FTSs. Consider a storage system with $N = 12$ servers, the size of FTS $R = 3$, and the scatter width $S = 4$. Using the BIBD-based method, the following solution of FTSs is created to achieve less number of FTSs. Using the BIBD-based method illustrated in Table 2.2, the following solution of FTSs is created.

$$B_1 = \{0, 1, 2\}, B_2 = \{3, 4, 10\}, B_3 = \{6, 7, 8\}, B_4 = \{9, 10, 11\},$$
$$B_5 = \{0, 3, 8\}, B_6 = \{1, 4, 7\}, B_7 = \{2, 5, 11\}, B_8 = \{5, 6, 9\}.$$

For random replication, the number of FTSs is 72. Hence, the probability of data loss caused by correlated machine failures is:

$$\#FTSs/\binom{N}{R} = 72/\binom{12}{3} = 0.327$$

However, for BIBD-based method from Copyset Replication, the number of FTSs is 8, and the probability of data loss caused by correlated machine failures is much smaller:

$$\#FTSs/\binom{N}{R} = 8/\binom{12}{3} = 0.036$$

There are many methods that can be used for constructing BIBDs, but no single method can create optimal BIBDs for any combination of N and R [64, 101]. Copyset Replication combines BIBD and random permutations to generate a non-optimal design that can accommodate any scatter width. By relaxing the constraint for the number of overlapping nodes λ from an exact number to at most the exact number, the probability of successfully generating FTSs increases. Since the scatter width should be much smaller than the number of nodes, MRR is likely to generate FTSs with at most one overlapping node [42]. MRR tries to minimize the replication degrees of data objects in order to

limit the consistency maintenance cost and storage cost, the replication degrees should be not large or vary greatly. Smaller replication degrees generate smaller R values. Although sometimes it is not possible to generate the optimal BIBDs, BIBDs can be used as a useful benchmark to measure how close MRR is to the optimal scheme for specific values of scatter width [42].

2.1.1.2 Uniform Machine Failures

In the scenario of uniform machine failures, the failures of machines are statistically independent of each other. Each machine has the same probability to fail, denoted by p ($0 < p < 1$). The data object is lost if any chunk (partition) of the data object is lost, and a chunk is lost only if all the replicas of the chunk are lost. Hence, the expected probability of data loss in the uniform machine failure is:

$$p_{uni} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} M \cdot p^{d_{ij}}}{m \cdot n} \tag{2.4}$$

where M is the number of partitions (chunks) for each data object, n is the number of applications, and m is the number of data objects in each application.

We introduce a method to evaluate a data object's popularity below. Different types of applications are always featured by the popular time periods of data objects. For example, the videos in social network applications are usually popular for 3 months [105–107], while the news in a news website usually is popular for several days. We rank the applications based on their types and use b_{a_i} to denote the rank; a higher b_{a_i} means that the application has longer popular time periods of data objects. Assume time is split into epochs (a fixed period of time). To avoid creating and deleting replicas for data objects that are popular for a short time period, we consider both its application rank (b_{a_i}) and its expected visit frequency, i.e., the number of visits in an epoch (v_{ij}) [18, 21, 28, 37], to determine the popularity function of a data object ($\varphi(\cdot)$):

$$\varphi_{ij}(\cdot) = \alpha \cdot b_{a_i} + \beta \cdot v_{ij} \tag{2.5}$$

where α and β are the weights for the two factors.

Let r_{ij} be the probability of requesting data D_{ij}. For single-object requests, $\sum_{i=1}^{n} \sum_{j=1}^{m} r_{ij} = 1$ after normalization. The request probability of a data object is the same as that of each partition of this data object. The request probability is proportional to the popularity of the data object

($\varphi(\cdot)$).

$$r_{ij} = k_1 \cdot \varphi_{ij}(\cdot) \tag{2.6}$$

where k_1 is a certain coefficient.

The popularity of a data object at epoch t is $\varphi_{ij}^t(\cdot)$. Then, the popularity of the data object at epoch $t+1$ can be calculated based on $\varphi_{ij}^t(\cdot)$:

$$\varphi_{ij}^{t+1}(b_{a_i}^{t+1}, v_{ij}^{t+1}) = \varphi_{ij}^t(b_{a_i}^t, v_{ij}^t) + \varphi_{ij}^t((b_{a_i}^{t+1} - b_{a_i}^t), (v_{ij}^{t+1} - v_{ij}^t)) \tag{2.7}$$

Hence, the request probability at epoch $t+1$ can be estimated at epoch t based on Formulas (2.5), (2.6) and (2.7). The cloud can create or delete the replicas at epoch t for epoch $t+1$ according to the calculated replication degree based on the estimated request probability r_{ij} at epoch t.

2.1.1.3 Nonuniform Machine Failures

Machines in a storage system may experience nonuniform failure rates due to different hardware/software compositions and configurations. Assume replicas of each data object are placed on machines with no concern for individual machine failures. Let $p_1, ..., p_N$ be the failure probabilities of N servers in the cloud, respectively. Based on the work in [139], the expected data object failure probability is the same as that on uniform failure machines with per-machine failure probability equaling $\frac{\sum_{i=1}^{N} p_i}{N}$. We use p_{non} to denote the expected probability of data loss in nonuniform machine failures. Then,

$$p_{non} = (\sum_{i=1}^{n} \sum_{j=1}^{m} M \cdot (\sum_{k=1}^{N} p_k/N)^{d_{ij}})/(m \cdot n) \tag{2.8}$$

2.1.1.4 Availability Maximization Problem Formulation

In a cloud system with the co-existence of correlated and non-correlated (uniform and nonuniform) machine failures, the data will be lost if any type of failures occurs. Recall that p_{cor} (Formula (2.2)), p_{uni} (Formula (2.4)) and p_{non} (Formula (2.8)) are the probabilities of a data object loss caused by correlated machine failures, uniform machine failures, and nonuniform machine failures (at epoch t), respectively. Hence, the probability of data loss caused by correlated and

non-correlated machine failures is

$$P_f = w_1 \cdot p_{cor} + w_2 \cdot p_{uni} + w_3 \cdot p_{non} \quad (\sum_{i=1}^{3} w_i = 1) \tag{2.9}$$

where w_1, w_2, and w_3 are the probabilities of the occurrence of each type of machine failures, respectively.

We maximize the data availability by minimizing the expected request failure probability with the consideration that different data objects have different popularities. To achieve this goal, we present a nonlinear programming model with multiple constraints to determine the replication degree of each data object.

Given the data object popularities and data object sizes, our goal is to find the optimal replication degree of each data object such that the expected request failure probability (denoted by $\bar{P}_r$) can be minimized. Each application purchases a certain storage space in public clouds. Also, in private clouds, different applications may be assigned different storage spaces based on their priorities. Thus, the storage space constraint for each application is necessary and important due to the limited precious storage mediums. Thus, we formalize our problem as the following constrained optimization problem:

$$\begin{aligned} min \quad & \bar{P}_r = \sum_{i=1}^{n} \sum_{j=1}^{m} r_{ij} \cdot M \cdot (P_f)^{d_{ij}} \\ s.t. \quad & \sum_{j=1}^{m} s_{ij} \cdot d_{ij} \leq K_i \ (i = 1, ..., n) \end{aligned} \tag{2.10}$$

where K_i $(i = 1, ..., n)$ is the total space capacity for application i, and s_{ij} is the size of data object D_{ij}. $\sum_{j=1}^{m} s_{ij} \cdot d_{ij}$ is the total storage consumption of all the data objects belonging to application i. The optimization objective is to minimize the expected request failure probability and the constraint is to ensure that the storage consumption of an application does not exceed its space capacity.

2.1.2 Consistency Maintenance Cost Minimization

In this section, we formulate the problem of minimizing consistency maintenance cost caused by data replication. We use C_c to denote the total consistency maintenance cost of all replicas in the cloud storage system. We first introduce how to calculate C_c. Previous work [68] indicates that the

data object write overhead is linear with the number of data object replicas. Then, the consistency maintenance cost of a partition can be approximated as the product of the number of replicas of the partition and the average communication cost parameter (denoted by δ_{com}) [28], i.e., $d_{ij} \cdot \delta_{com}$. Hence, the total consistency maintenance cost of all data objects is

$$C_c = \sum_{i=1}^{n} \sum_{j=1}^{m} (M \cdot d_{ij}) \cdot \delta_{com} \tag{2.11}$$

The fixed average communication cost (δ_{com}) can be calculated as in [128]:

$$\delta_{com} = E[\sum_{i,j} s^u \cdot dis(S_i, S_j) \cdot \sigma] \tag{2.12}$$

where s^u is the average update message size, $dis(S_i, S_j)$ is the geographic distance between the server of the original copy S_i and a replica server S_j, and σ is the average communication cost per unit of distance. We adopt the method in [28] to calculate the geographic distance between servers. We present the details of this method below.

This method uses a 6-bit number to represent the distance between servers. Each bit corresponds to the location part of a server, i.e., continent, country, datacenter, room, rack and server. Starting with the most significant bit (i.e., the leftmost bit), each location part of both servers are compared one by one to compute the geo-similarity between them. The corresponding bit is set to 1 if the location parts are equivalent, otherwise it is set to 0. Once a bit is set to 0, all less significant bits are automatically set to 0. For example, suppose S_i and S_j are two arbitrary servers, and the distance between them is represented as 111000 (as shown below). Then, it indicates that S_i and S_j are in the same datacenter but not in the same room.

continent	*country*	*datacenter*	*room*	*rack*	*server*
1	1	1	0	0	0

The geographic distance is obtained by applying a binary "NOT" operation to the geo-similarity. That is,

$$\overline{111000} = 000111 = 7 \ (decimal)$$

Thus, the optimization problem for consistency maintenance cost can be formulated as follows:

$$\begin{aligned} min \quad & C_c = \sum_{i=1}^{n}\sum_{j=1}^{m}(M \cdot d_{ij}) \cdot \delta_{com} \\ s.t. \quad & \sum_{j=1}^{m} s_{ij} \cdot d_{ij} \leq K_i \ (i = 1, ..., n) \end{aligned} \tag{2.13}$$

2.1.3 Storage Cost Minimization

Different applications require different storage mediums (e.g., disk, SSD, EBS) to store data. Different mediums have different costs per unit size. For example, in the Amazon web cloud service, the prices for EBS and SSD are \$0.044 and \$0.070 per hour, respectively. After satisfying the applications' storage requirements on different storage mediums, we need to decide the storage mediums for their additional replicas for enhanced data availability. Different applications have different SLAs with different associated penalties. The applications with higher SLA violation penalties should have higher priorities to meet their SLA requirements in order to reduce the associated penalties. Therefore, the additional replicas of data objects of higher-priority applications should be stored in a faster storage medium. On the other hand, in order to save storage cost, the additional replicas of data objects of lower-priority applications should be stored in a lower and less expensive storage medium. We use b_{p_i} to denote the priority of application i; higher b_{p_i} means higher priority. Since faster storage mediums are more costly, for data objects of high priority applications, we hope to store more data partitions in faster storage mediums in order to satisfy more requests per unit time hence increase data availability [28]. As a result, to determine the storage medium used for storing additional replicas, we define a priority function $\psi(\cdot)$ of a data object based on its application priority and its size:

$$\psi_{ij}(\cdot) = \gamma \cdot b_{p_i} + \eta / s_{ij} \tag{2.14}$$

where γ and η are the weights for application priority and the size of the data object. The size of the data object can be changed due to write operation. The priority values are classified to a number of levels, and each level corresponds to a storage medium. Thus, the unit storage cost of a data object equals the unit cost of its mapped storage medium, denoted by $c_{s_{ij}}$, which is proportional to the

priority value of a data object ($\psi_{ij}(\cdot)$).

$$c_{s_{ij}} = k_2 \cdot \psi_{ij}(\cdot) \tag{2.15}$$

where k_2 is a certain coefficient.

The priority value of a data object at epoch t is $\psi_{ij}^t(\cdot)$. Hence, the priority value of the data object at epoch $t+1$ can be estimated based on the priority value at epoch t:

$$\psi_{ij}^{t+1}(b_{p_i}^{t+1}, s_{ij}^{t+1}) = \psi_{ij}^t(b_{p_i}^t, s_{ij}^t) + \psi_{ij}^t((b_{p_i}^{t+1} - b_{p_i}^t), (s_{ij}^{t+1} - s_{ij}^t)) \tag{2.16}$$

Hence, the unit storage cost of a data object at epoch $t+1$ can be estimated at epoch t based on Formulas (2.14), (2.15) and (2.16). The cloud can determine the storage mediums for data objects at epoch t for epoch $t+1$ based on the estimated unit storage cost at epoch t.

The storage cost of a data object is related to its storage medium, its size and the number of its replicas. Different storage mediums have different unit costs, and different data objects have different sizes and replication degrees. We minimize storage cost by minimizing the expected cost for storage mediums. We examine expected storage cost minimization by determining the replication degree for each data object. To achieve this goal, we present a nonlinear programming approach with multiple constraints that can be used to obtain a policy.

Given the applications of data objects and the data object sizes, our goal is to find the optimal replication degree for each data object that minimizes storage cost. Hence, we formalize our problem as the following constrained optimization problem:

$$\begin{aligned} min \quad & C_s = \sum_{i=1}^{n}\sum_{j=1}^{m}(s_{ij} \cdot d_{ij} \cdot c_{s_{ij}} \cdot T) \\ s.t. \quad & \sum_{j=1}^{m} s_{ij} \cdot d_{ij} \leq K_i \ (i = 1, ..., n) \end{aligned} \tag{2.17}$$

where T is the time duration of an epoch.

2.1.4 Problem Formulation and Solution

We consider additional three threshold constraints for request failure probability (P_r^{th}), consistency maintenance cost (C_c^{th}) and storage cost (C_s^{th}). The probability of expected request

failure must be no larger than a threshold P_r^{th}. This constraint is used to ensure that the expected request failure probability is not beyond a threshold, which serves the goal of achieving high data availability. The constraint on the consistency maintenance cost is to ensure that the consistency maintenance cost in one epoch is no larger than a threshold, C_c^{th}. The storage cost in one epoch is no more than threshold C_s^{th}. The constraints on both consistency maintenance cost and storage cost are to ensure that the cost caused by replication is at a low level, which makes the system more efficient and economical. By combining Formulas (2.10), (2.13) and (2.17) and the additional constraints, we can build a nonlinear programming model (NLP) for the global optimization problem as follows:

$$\begin{aligned} min \ \ \{\bar{P}_r + C_c + C_s\} = & \sum_{i=1}^{n}\sum_{j=1}^{m}(r_{ij} \cdot M \cdot (P_f)^{d_{ij}}) \\ & + \sum_{i=1}^{n}\sum_{j=1}^{m}(M \cdot d_{ij}) \cdot \delta_{com} + \sum_{i=1}^{n}\sum_{j=1}^{m}(s_{ij} \cdot d_{ij} \cdot c_{s_{ij}} \cdot T) \end{aligned} \tag{2.18}$$

$$s.t. \ \ \sum_{j=1}^{m} s_{ij} \cdot d_{ij} \leq K_i \ (i = 1, ..., n) \tag{2.19}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m} r_{ij} \cdot M \cdot (P_f)^{d_{ij}} \leq P_r^{th} \tag{2.20}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}(M \cdot d_{ij}) \cdot \delta_{com} \leq C_c^{th} \tag{2.21}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}(s_{ij} \cdot d_{ij} \cdot c_{s_{ij}} \cdot T) \leq C_s^{th} \tag{2.22}$$

The objective is to minimize the request failure probability, the consistency maintenance cost and storage cost. The optimization constraints are used to ensure that the space capacity of data objects belonging to each application is not exceeded, the probability of expected request failure is no more than a threshold P_r^{th}, the consistency maintenance cost is no more than threshold C_c^{th}, and the

storage cost in one epoch is no more than threshold C_s^{th}.

Theorem 3.1. *The relaxed NLIP optimization model is convex.*

Proof Inequs. (2.19), (2.21), (2.22) are linear inequalities, and they define convex regions. The exponential function $P_f^{d_{ij}}$ in Inequ. (2.20) is convex, and the sum (i.e., $\sum_{i=1}^{n}\sum_{j=1}^{m} r_{ij}\cdot m\cdot(P_f)^{d_{ij}}$) of convex functions is a convex function. Thus, the constraint (2.20) defines a convex set. All the constraints define convex regions, and the intersection of convex sets is a convex set. Thus, the region of the optimization problem is convex. Hence the relaxed NLIP optimization model is convex.

Therefore, Theorem 3.1 holds. ■

We then solve this optimization problem. For analytical tractability, we first relax the problem to a real-number optimization problem in which $d_{11}, ..., d_{1m}, ..., d_{n1}, ..., d_{nm}$ are real numbers, and derive the solution for the real-number optimization problem. Then, we use integer rounding to get the solution for practical use. Specifically, we adopt the approach from [139] to round each d_{ij} to its nearest integer while all d_{ij}'s smaller than 1 are rounded to 1. By relaxing the problem to real-number optimization problem, the optimal solution should always use up all the available storage space (i.e., K_i) for each application [139].[2] Thus, we have

$$\sum_{j=1}^{m} s_{ij} \cdot d_{ij} = K_i \quad (i = 1, ..., n) \tag{2.23}$$

where K_i is the space capacity for application i. For the real-number optimization problem, we use Lagrange multipliers to derive the solution. Since there are $n+3$ constraints, we use the multipliers $\lambda_1, \lambda_2, ..., \lambda_{n+3}$ to combine the constraints and the optimization goal together into the Lagrangian

[2]The storage cost may increase as storage space increases, but it also depends on the storage media for storing the data, thus the claim that the space constraint holds at equality does not contradict the main hypothesis (cost-effective) of the proposed approach.

function

$$
\begin{aligned}
&\Lambda(d_{11}, ..., d_{1m}, ..., d_{n1}, ..., d_{nm}, \lambda_1, \lambda_2, ..., \lambda_{n+3}) \\
=&\sum_{i=1}^{n}(\sum_{j=1}^{m} r_{ij} \cdot M \cdot (P_f)^{d_{ij}} + \sum_{j=1}^{m}(M \cdot d_{ij}) \cdot \delta_{com} \\
&+ \sum_{j=1}^{m}(s_{ij} \cdot d_{ij} \cdot c_{s_{ij}} \cdot T) + \lambda_i(\sum_{j=1}^{m} s_{ij} \cdot d_{ij} - K_i)) \\
&+ \lambda_{n+1}(\sum_{i=1}^{n}\sum_{j=1}^{m} r_{ij} \cdot M \cdot (P_f)^{d_{ij}} - P_r^{th}) \\
&+ \lambda_{n+2}(\sum_{i=1}^{n}\sum_{j=1}^{m}(M \cdot d_{ij}) \cdot \delta_{com} - C_c^{th}) \\
&+ \lambda_{n+3}(\sum_{i=1}^{n}\sum_{j=1}^{m}(s_{ij} \cdot d_{ij} \cdot c_{s_{ij}} \cdot T) - C_s^{th})
\end{aligned}
\tag{2.24}
$$

where $K_i = \sum_{j=1}^{m} s_{ij} \cdot d_{ij}$. The critical values of Λ is achieved only if its gradient is zero. Based on Theorem 3.1, the NLP optimization problem is convex. Thus, the gap between the relaxed problem and its dual problem is zero [24]. Also, the object function of the NLP model is derivable, thus the gradients of λ exist. We can get the solution for the optimization problem based on the Lagrange dual solution. Considering that the popularities and importance of data objects usually do not change much within a short period of time, we can choose larger value for T to avoid computation overhead, and also use IPOPT [8, 124] (a software library for large scale nonlinear optimization) to solve the large-scale nonlinear optimization problem, which make MRR more practical.

2.2 The MRR Replication Scheme

In Section 2.1.4, we calculate the optimal replication degree of each data object so that the request failure probability, consistency maintenance cost and storage cost are minimized in a cloud system with both correlated and non-correlated machine failures. The next problem is how to assign the replicas to nodes, which is of importance for increasing the data availability [58]. Recall that in Section 2.1.1, we introduced BIBD-based file replication in Copyset Replication that can handle correlated machine failures. Though the BIBD-based method can be used for replication to handle correlated machine failures, it requires a constant replication degree and cannot be used for replicating data with various replication degrees of different data objects. Our proposed MRR can

deal with the problems. We present the details of MRR below.

Algorithm 1: Pseudocode of the MRR algorithm.

```
1 Compute the replication degree for each data object using the NLP model (Section
  2.1.4)
2 Rank the replication degrees in ascending order d_1, ..., d_l
3 for i ← 1 to l do
4     Group data objects with d_i together (D_i)
5     Use BIBD-based method to generate FTSs; each FTS has nodes from different
      datacenters but within a certain geographic distance
6     Store each chunk's replicas of data objects with d_i to all nodes in an FTS with d_i
7 return
```

Algorithm 1 shows the pseudocode of the MRR replication algorithm. For particular and arbitrary i and j, the replication degree d_{ij} of data D_{ij} can be obtained from the NLP optimization model in Section 2.1.4. To reduce data loss in both correlated and non-correlated machine failures, MRR first ranks these replication degrees in ascending order (i.e., $d_1, d_2, ..., d_l$). For a given replication degree d_i ($i = 1, 2, ..., l$), MRR first groups the data objects with replication degree d_i together (denoted by D_i), and counts the number of data objects with replication degree d_i (denoted by $N^r_{D_i}$). To handle the problem of varying replication degrees, MRR partitions all nodes to l groups and then conducts Copyset Replication [42] to assign chunks with replication degree d_i to the ith node group ($i = 1, 2, ..., l$). For load balance, the number of nodes in each group is proportional to the number of replicas that will be stored in the group. Accordingly, MRR assigns $|N \cdot \frac{N^r_{D_i} \cdot d_i}{\sum_{i=1}^{l} N^r_{D_i} \cdot d_i}|$ nodes to data group D_i. MRR then uses the BIBD-based method to generate FTSs for each group of nodes. Specifically, MRR determines λ in Section 2.1.1.1 based on the load balancing requirement and then uses the BIBD-based method for (N, R, λ), where N is the number of the nodes in a node group and R is the replication degree of the group (d_i). The nodes in each FTS are required to be from different datacenters and within a certain geographic distance between each other. Distributing replicas over different datacenters can avoid data loss due to machine failures (e.g., caused by power outages) for data reliability [1]. Limiting the distances between replica nodes for a data chunk constrains the consistency maintenance cost. MRR replicates the chunks of each data object to all nodes in an FTS.[3] Figure 2.1 illustrates an example to show the design of MRR. In the example, $d_1 = 2$, $d_2 = 3$, and $d_3 = 4$. In the left most block, each FTS contains two chunks of a data object. Each FTS

[3] Although putting all replicas of a chunk to the nodes in an FTS can bring about the cost of inter-rack transfer (across oversubscribed switches), it can significantly reduce the probability of data loss caused by correlated machine failures by using BIBD-based method [42].

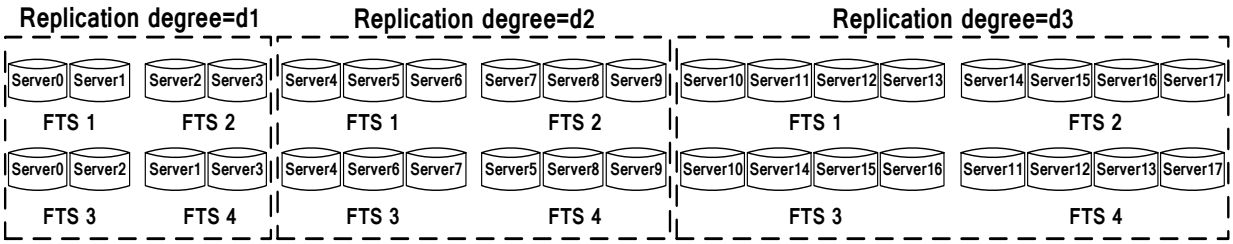


Figure 2.1: Architecture of MRR for cloud storage fault-tolerant sets (FTSs).

overlaps with every other FTS with at most one server. This helps reduce the probability of data loss in correlated machine failures and balance the load. For those data objects with replication degree 2, the chunks of a data object will be stored to the nodes in an FTS in the left most block. In the middle block, each FTS contains three chunks of a data object. In the right most block, each FTS contains four chunks of a data object.

Recall that each data priority value corresponds to a storage medium. We assume that all servers have all types of storage mediums, and we need to determine which medium to use for a given priority on a given server. When replicating a chunk to a node, MRR chooses the storage mediums for the chunks based on data objects' priority calculated by Formula (2.14). Data objects with higher priority values will be stored to faster and more expensive storage mediums (e.g., Memory, SSD), and data objects with lower priority value will be stored to slower and cheaper storage mediums (e.g., disk). The constraints (2.19) is to ensure that storage requirements of data objects do not overfill the servers.

The design of MRR can reduce the probability of data loss in both correlated and non-correlated machine failures. MRR uses the BIBD-based method to reduce the probability of data loss in correlated machine failures by minimizing the number of FTSs, and uses the replication degree determined by NLP (in Section 2.1.4) with multiple constraints to maximize data availability in correlated and non-correlated machine failures and minimize the cost caused by replication.

2.3 Performance Evaluation

We conducted the numerical analysis based on the parameters in [42] (Table 2.3) derived from the system statistics from Facebook, HDFS and RAMCloud [11, 30, 35, 42, 87, 111], and also conducted real-world experiments on Amazon S3. The distributions of file read and write rates in our analysis and tests follow those of the CTH trace [85], which is provided by Sandia National

Table 2.3: Parameters from publicly available data [42].

System	Chunks per node	Cluster size	Scatter width
Facebook	10000	1000-5000	10
HDFS	10000	100-10000	200
RAMCloud	8000	100-10000	N-1

Laboratories that records 4-hour read/write log in a parallel file system. In the following, we introduce our numerical analysis results and real-world experimental results, respectively.

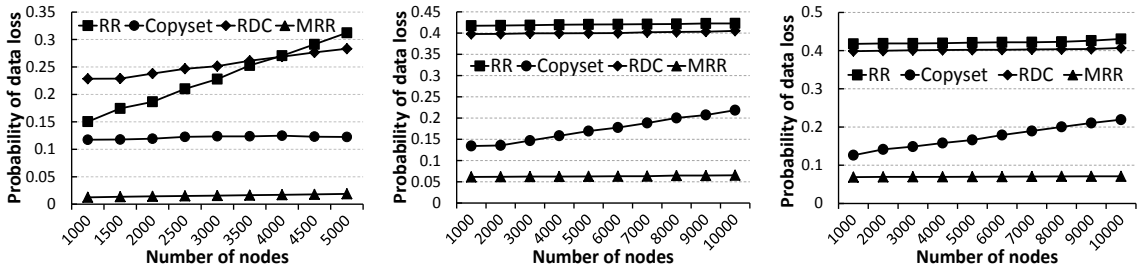
2.3.1 Numerical Analysis Results

We conducted numerical analysis under various scenarios. We compared our proposed replication scheme MRR with other three replication schemes: Random Replication (RR) [42], Replication Degree Customization (RDC) [139] and Copyset Replication [42] (Copyset). RR places the primary replica on a random node (say node i) in the entire system, and places the secondary replicas on (R-1) nodes around the primary node (i.e., nodes i+1, i+2,...).[4] RDC derives replication degree for each object with the consideration of data object popularity to maximize the expected data availability for non-correlated machine failures. Copyset splits the nodes into a number of copysets, and constrains the replicas of every chunk to a single copyset so that it can reduce the frequency of data loss by minimizing the number of copysets for correlated machine failures. The number of nodes that fail concurrently in each system was set to 1% of the nodes in the system [42, 99]. Since this rate is the maximum percentage of concurrent failure nodes in real clouds (i.e., worst case) [35, 44, 53], it is reasonable to see higher probabilities of data loss and lower expected data availability in our analytical results than the real results in current clouds. The distributions of file popularity and updates follow those of the CTH trace. We used the normal distribution with mean of 10 and standard deviation of 1 to generate 10 unit costs for different storage mediums. Compared to uniform machine failures, nonuniform and correlated machine failures are more realistic due to different hardware/software compositions and configurations [139]. Thus, we set $w_1 = 0.4$, $w_2 = 0.2$ and $w_3 = 0.4$ in Equ. (2.9), respectively. We randomly generated 6 bit number from reasonable ranges for each node to represent its location, as explained in Section 2.1.2. Table 4.2 shows the parameter settings in our analysis unless otherwise specified.

We first calculate the probability of data loss for each method.[5] Specifically, we used For-

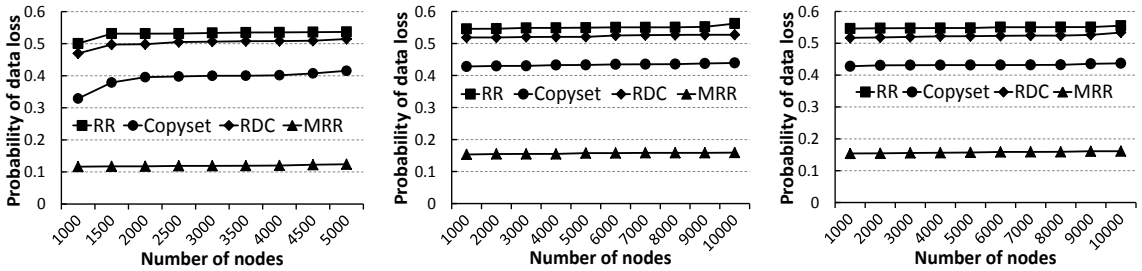
[4]RR is based on Facebook's design, which chooses secondary replica holders from a window of nodes around the primary node.

[5]Many datacenter operators prefer to low probability of any incurring data loss at the expense of losing more data



(a) Facebook (b) HDFS (c) RAMCloud

Figure 2.2: Probability of data loss vs. number of nodes ($R = 3$ for RR and Copyset).



(a) Facebook (b) HDFS (c) RAMCloud

Figure 2.3: Probability of data loss vs. number of nodes ($R = 2$ for RR and Copyset).

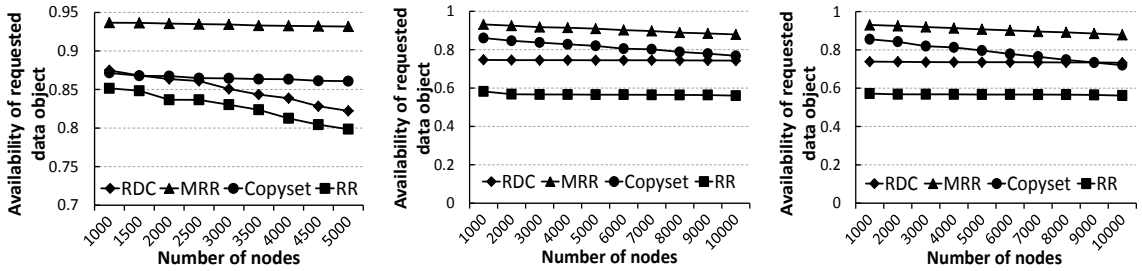
mula (2.9) for MRR, Formula (2.2) for Copyset, Formula (2.3) for RR, and Equ. (2.9) with $w_1 = 0$, $w_2 = 0.4$ and $w_3 = 0.6$ for RDC. Figure 2.2(a), 2.2(b) and 2.2(c) show the relationship between the probability of data loss and the number of nodes in the Facebook, HDFS and RAMCloud environments, respectively. We find that the result approximately follows MRR<Copyset<RDC<RR. The probability of data loss in Copyset is higher than that in MRR. This is because MRR considers non-correlated machine failures which are not considered in Copyset. Specifically, MRR considers the failure probability of individual machines in determining replication degree based on data object popularity. RDC generates a higher probability of data loss than MRR and Copyset because it neglects reducing the probability of data loss caused by correlated machine failures which often occur in large-scale storage systems. The probability of data loss in RR is much higher than MRR and Copyset, and also higher than RDC. This is due to two reasons. First, RR places the copies of a chunk on a certain number (i.e., R) of different nodes. Any combination of R nodes that fail simultaneously would result in data loss in correlated machine failures. Unlike RR, MRR and Copyset split the nodes into FTSs consisting of a certain number (i.e., R) of nodes, and constrain the copies of a chunk into a single FTS. Then, they lose data only if all the nodes in an FTS fail simultaneously.

in each event due to high cost of each incident of data loss [42].

Second, MRR and RDC consider data popularity to increase the expected data availability, which however is not considered in RR. To further verify MRR's performance in reducing the probability of data loss, we changed the value of R from 3 to 2. Figure 2.3 shows the relationship between the probability of data loss and the number of nodes with $R = 2$. Figure 2.3 mirrors Figure 2.2 due to the same reasons. Comparing Figure 2.3 to Figure 2.2, we find that the probability of data loss decreases as the number of replicas R increases. This is because the more the replicas for a chunk, the lower the probability that all the machines storing the chunk fail simultaneously, and thus the lower the probability of the chunk being lost.

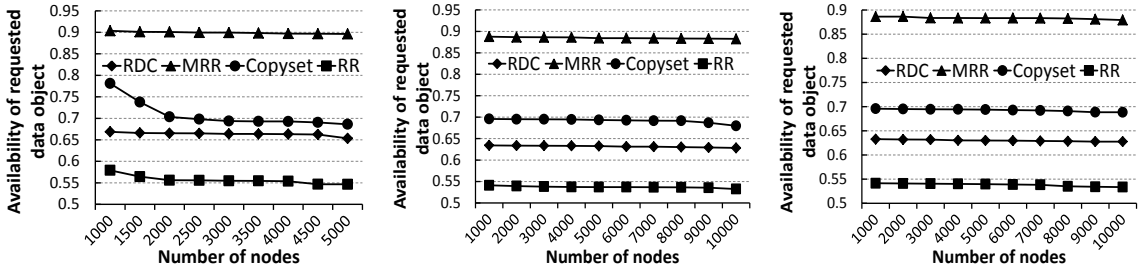
We then calculate the availability of requested data object by $1-\bar{P}_r$, and $\bar{P}_r$ is calculated by Formula (2.10). Figure 2.4(a), 2.4(b) and 2.4(c) show the relationship between the availability of requested data objects and the number of nodes in the Facebook, HDFS and RAMCloud environments, respectively. We see that in all figures, the result generally follows MRR>Copyset>RDC>RR. The availability of requested data objects in Copyset is lower than that in MRR due to two reasons. First, MRR considers data object popularity when determining the replication degree for each chunk, which however is not considered in Copyset. By creating more replicas for popular data objects and less replicas for unpopular data objects, MRR can satisfy the demands of data objects with different popularities and increase the overall expected availability of requested data objects. Second, MRR reduces data loss in both correlated and non-correlated machine failures, while Copyset only minimizes the data loss in correlated machine failures. The availability of requested data objects in Copyset is higher than that in RDC. This is due to the reason that RDC cannot reduce the probability of data loss caused by correlated machine failures and thereby increases the probability of data loss, which decreases the availability of requested data objects. The availability of requested data objects in RR is the lowest because RR places the copies of a chunk on a certain number (i.e., R) of nodes and any combination of R nodes that fail simultaneously would cause data loss. Also, it does not consider data popularity as RDC, which is important to increase the expected data availability.

We also varied R from 3 to 2 to better verify the availability of MRR. Figure 2.5 shows the relationship between the availability of requested data objects and the number of nodes with $R = 2$. Figure 2.5 mirrors Figure 2.4 due to the same reasons. Comparing Figure 2.5 with Figure 2.4, we find that the availability of requested data objects increases as the number of replicas R increases. This is because the more replicas for a data object, the lower the probability of the data object



(a) Facebook (b) HDFS (c) RAMCloud

Figure 2.4: Availability of requested data objects vs. number of nodes ($R = 3$ for RR and Copyset).



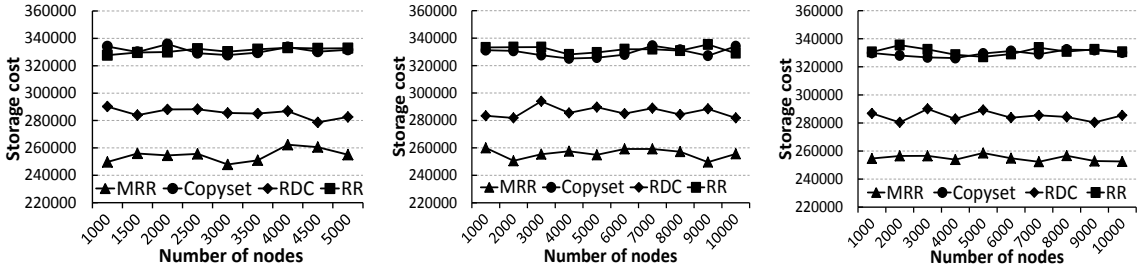
(a) Facebook (b) HDFS (c) RAMCloud

Figure 2.5: Availability of requested data objects vs. number of nodes ($R = 2$ for RR and Copyset).

being lost, thus the higher the availability of the requested data object.

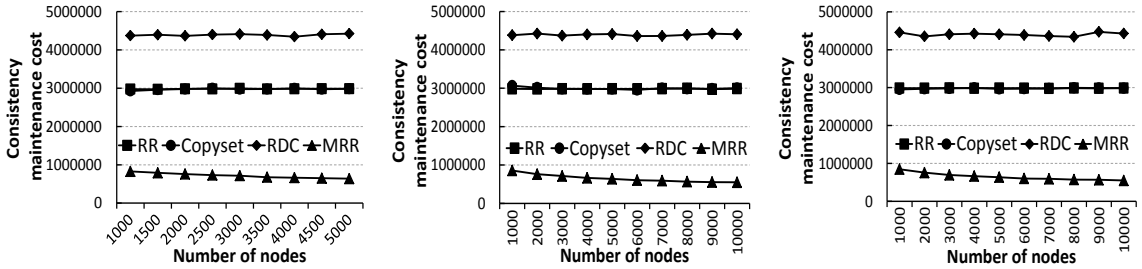
We then used Formula (2.17) to calculate the storage cost based on the sizes, replication degrees, and storage medium unit costs of data objects for MRR. For the other three methods, we randomly choose storage mediums for data objects and do not minimize the storage cost with space capacity constraint ($\sum_{i=1}^{n}\sum_{j=1}^{m} s_{ij} \cdot d_{ij} \leq \sum_{i=1}^{n} K_i$) for RDC. Figure 2.6(a), 2.6(b) and 2.6(c) show the relationship between the storage cost and the number of nodes in the Facebook, HDFS and RAMCloud environments, respectively. The result in all three figures generally follows RR≈Copyset>RDC>MRR. The storage cost in RDC is higher than that in MRR. This is because MRR stores data objects into different storage mediums based on the priority of their applications, the sizes, and the replication degrees of data objects to minimize the total storage cost. The storage costs in Copyset and RR are higher than RDC and much higher than MRR. RDC reduces the replicas of unpopular data objects, thus reducing storage cost. Copyset and RR neither consider the different storage mediums nor reduce the replicas of unpopular data objects to reduce storage cost. These results indicate the lower storage cost of MRR by considering both the data popularity and storage medium cost in replication. Therefore, the storage costs in Copyset and RR are higher than

that in RDC and much higher than that in MRR.



(a) Facebook (b) HDFS (c) RAMCloud

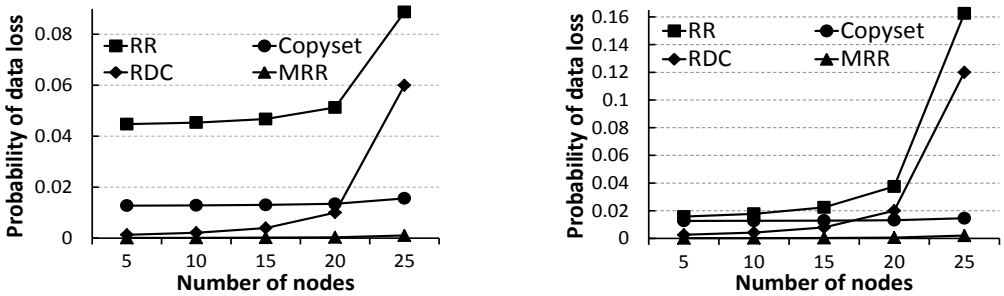
Figure 2.6: Storage cost vs. number of nodes ($R = 3$ for RR and Copyset).



(a) Facebook (b) HDFS (c) RAMCloud

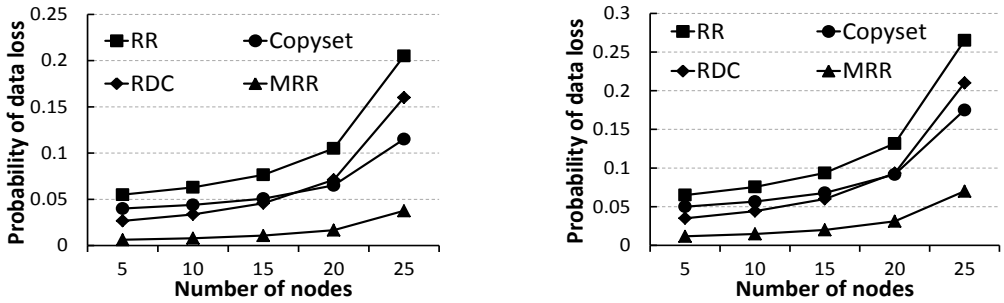
Figure 2.7: Consistency maintenance cost vs. number of nodes ($R = 3$ for RR and Copyset).

We used Formula (2.13) to calculate the consistency maintenance cost of each method based on the geographic distance and replication degrees of data objects. Figure 2.7(a), 2.7(b) and 2.7(c) show the relationship between the consistency maintenance cost and the number of nodes in the Facebook, HDFS and RAMCloud environments, respectively. In these figures, the results of Copyset and RR are overlapped and MRR generates the lowest consistency maintenance cost. The consistency maintenance costs in Copyset and RR are higher than MRR because MRR limits the geographic distance between the replica nodes of a chunk and the number of replicas, thereby reducing the consistency maintenance cost. However, Copyset and RR neglect geographic distances. RDC produces higher consistency maintenance cost than Copyset and RR. RDC neglects geographic distance and it also generates more replicas for popular data objects. These results indicate MRR generates much lower consistency maintenance cost than other methods.



(a) Scatter width=2 (b) Scatter width=4

Figure 2.8: Probability of data loss vs. number of nodes on Amazon S3 ($R = 3$ for RR and Copyset).



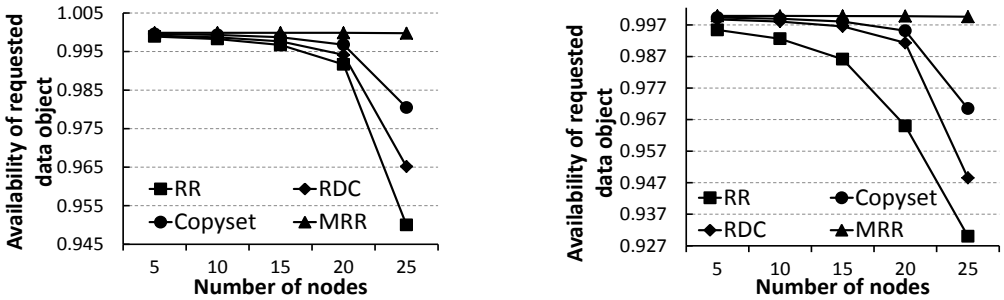
(a) Scatter width=2 (b) Scatter width=4

Figure 2.9: Probability of data loss vs. number of nodes on Amazon S3 ($R = 2$ for RR and Copyset).

2.3.2 Real-world Experimental Results

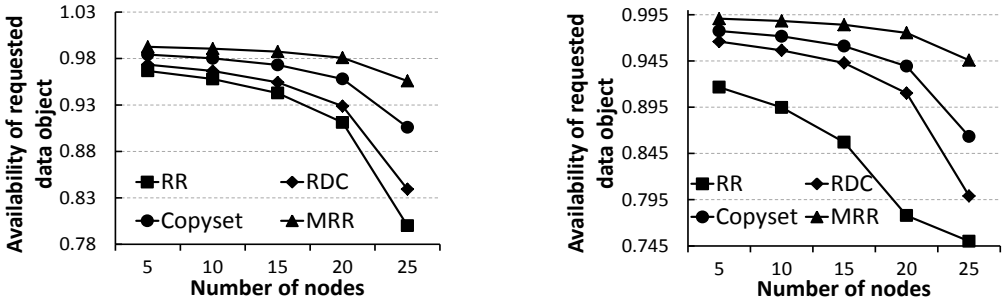
We conducted experiments on the real-world Amazon S3. We simulated the geo-distributed storage datacenters using three regions of Amazon S3 in the U.S. In each region, we created the same number of buckets, each of which simulates a data server. The number of buckets was varied from 5 to 25 with step size of 5 in our test. We distributed 5000 data objects to all the servers in the system. We used the distributions of read and write rates from the CTH trace data to generate reads and writes. The requests were generated from servers in Windows Azure eastern region. We consider the requests targeting each region with latency more than 100ms as failed requests (unavailable data objects).[6] We used the parameters in Table 4.2 except p and N. In this test, N is the total number of simulated data servers in the system and p follows the actual probability of a server failure in the real system. We used the actual price of the data access of Amazon S3 to calculate the storage cost.

[6]Since it is hard to generate permanent failures on Amazon S3 and the network latency is low based on [6], we think the failure of data request within U.S. is mainly caused by machine failures, and we consider the requests targeting each region with latency longer than 100ms as failed request, which reflects the availability of data objects.



(a) Scatter width=2 (b) Scatter width=4

Figure 2.10: Availability of requested data objects vs. number of nodes on Amazon S3 ($R = 3$ for RR and Copyset).

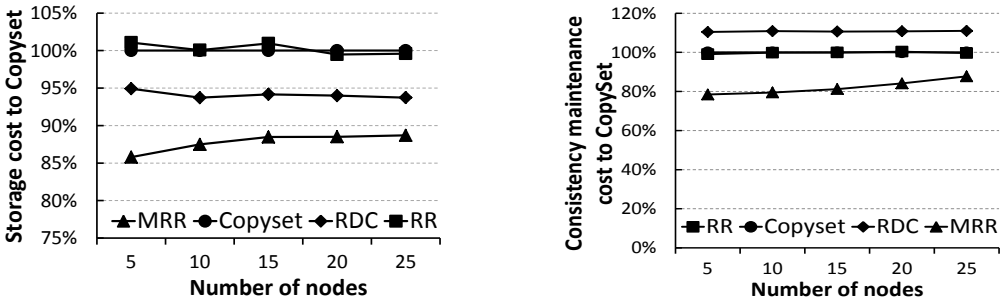


(a) Scatter width=2 (b) Scatter width=4

Figure 2.11: Availability of requested data objects vs. number of nodes on Amazon S3 ($R = 2$ for RR and Copyset).

Figure 2.8(a) and 2.8(b) show the relationship between the probability of data loss and the number of nodes on Amazon S3 when the scatter width (S) equals to 2 and 4, respectively. We see that the result approximately follows MRR<RDC<Copyset<RR. Our numerical result shows that MRR<Copyset<RDC<RR. Both results confirm that MRR generates the lowest probability of data loss. RDC generates higher probability of data loss than Copyset in the numerical analysis but generates lower probability of data loss than Copyset in the experiments. This is because RDC cannot handle correlated machine failures, and the failure rate of correlated machine failures is 1% in the numerical analysis but our real-world experiment has fewer correlated machine failures. As a result, RDC generates a relatively higher probability of data loss in the numerical analysis. We also see that scatter width 2 produces lower probability of data loss than scatter width 4. This is because a large scatter width increases the number of FTSs and thus increases the probability of data loss.

To further verify the MRR's performance in reducing data loss probability, we then decreased



(a) Storage cost

(b) Consistency maintenance cost

Figure 2.12: Storage cost and consistency maintenance cost vs. number of nodes on Amazon S3 ($R = 3$ for RR and Copyset).

the value of R to 2 in Figure 2.9. Figure 2.9 shows the relationship between the probability of data loss and the number of nodes on Amazon S3 with $R = 2$. Figure 2.9 mirrors Figure 2.8 due to the same reasons. Comparing Figure 2.9 with Figure 2.8, we find that the probability of data loss decreases as R increases. This is because the larger the number of replicas for a chunk, the lower the probability that all the machines storing the chunk fail concurrently, and thus the lower the probability that the chunk is lost.

Figure 2.10(a) and 2.10(b) show the relationship between the availability of requested data objects and the number of nodes on Amazon S3 when the scatter width equals to 2 and 4, respectively. We see that the result follows MRR>Copyset>RDC>RR, which is consistent with the result in Figure 2.4 due to the same reasons. We also see that scatter width 2 produces higher availability than scatter width 4. This is because a large scatter width increases the number of FTSs and thus increases the probability of data loss and reduces the data availability. Therefore, it is important to choose an appropriate scatter width to achieve a tradeoff between load balance and data availability as mentioned in Section 2.1.1

We also decreased R to 2 in Figure 2.11. Figure 2.11 mirrors Figure 2.10 due to the same reasons. Comparing Figure 2.11 with Figure 2.10, we find that the availability of requested data objects increases as R increases due to the similar reasons explained in the comparison of Figure 2.9 and 2.8.

We then regard Copyset as a baseline and calculate the ratio of the storage cost of each of the other methods over Copyset's storage cost. Figure 2.12(a) shows the storage cost ratio of different schemes. We see that the result follows RR≈Copyset>RDC>MRR, which is consistent with that in Figure 2.6 due to the same reasons. Figure 2.12(b) shows the ratio of the consistency

maintenance cost of each method over Copyset's consistency maintenance cost. We see that the result follows RDC>RR≈Copyset>MRR, which is consistent with that in Figure 2.7 due to the same reasons.

2.4 Conclusion

This chapter presents a low-cost multi-failure resilient replication scheme (MRR) for high data availability in cloud storage. Specifically, the chapter first introduces an NLIP model for MRR, which formulates a problem that determines the replication degree of each data object so that the request failure probability, consistency maintenance cost and storage cost are minimized in both correlated and non-correlated machine failures. Then, the chapter describes the design of MRR based on the problem solution derived from the NLIP model. Finally, the chapter describes the numerical analysis and real-word experiments on Amazon S3, which show that MRR outperforms other replication schemes in different performance metrics.

Chapter 3

CORP: Cooperative Opportunistic Resource Provisioning for Short-Lived Jobs in Cloud Systems

In this chapter, we introduce our cooperative opportunistic resource provisioning scheme (CORP) for short-lived jobs in cloud systems. We first introduce the background of deep learning. Next, we describe the cooperative opportunistic resource provisioning problem in clouds. Then, we present the design of CORP. Experimental results based on a real cluster and Amazon EC2 show that CORP achieves high resource utilization and low SLO violation rate compared to previous resource provisioning schemes.

3.1 Background

In this section, we review the background of deep learning. Artificial neural networks consist of a lot of homogeneous computing units called neurons with multiple inputs and a single output. These are typically connected in a layer-wise manner with the output of neurons in a layer connected to all neurons in the next upper layer (see Figure 1.5). Deep neural networks have multiple layers, which can enable hierarchical feature learning and gives the potential of modeling complex data with fewer units than a similarly performing shallow network [25]. The output of a neuron

in a layer, called the activation, is computed as a function of its input. The activation function associated with all neurons in the network is a predefined non-linear function. Typically, a sigmoid or hyperbolic tangent is used for activation function. Deep neural networks are typically trained by back propagation using gradient descent. Stochastic gradient, a variant of gradient descent, is always used for scalable training in that it requires less cross-machine communication [31].

Compared with other prediction methods, deep learning only needs the raw data for training without requiring sufficient high quality and truly representative past data being available [72]. Also, compared to other machine learning methods (e.g., Support Vector Machine (SVM), Bayesian approaches, Decision Trees, etc.), deep learning has better accuracy in many applications [76, 112, 134, 136], and it can more accurately predict the real resource demands with the same given past data. It achieves high accuracy by multiple training epoches. It reprocesses the training data set each time until the validation set error converges to a desired value. In addition, deep neural networks with multiple hidden layers are capable of modeling complex data with greater efficiency and they are shown to have an advantage over shallow machine methods [39]. Finally, deep learning does not require the existence of patterns in the historical data, which is required by the methods such as fast Fourier transform [34, 108]. Deep neural networks has been successfully applied to fields such as language modeling, vision and automatic speech recognition.

3.2 Cooperative Opportunistic Resource Provisioning Problem

The physical machines (PMs) are deployed in a cloud system, and their resources are allocated to virtual machines (VMs). The VM capacity comprises of multiple types of resource (e.g., CPU, MEM and storage) and their resources are allocated to jobs based on job workloads. In this dissertation, we consider the problem of allocating allocated but unused resources in VMs to jobs for achieving high resource utilization and low SLO violation rate. To increase the resource utilization, the allocated but unused resource can be reallocated to jobs with a certain probability. Also, for jobs with different resource intensities (e.g., a job with high demand on CPU and a job with a high demand on MEM), they can be allocated with the unused resources in a VM together to reduce resource fragmentation in order to further increase resource utilization.

Suppose there are N_v VMs, and l types of resources (e.g., CPU, MEM, storage) in the

system. We use v_i to denote the i-th VM and use C_{ij} to denote the capacity for the type j resource of VM v_i. Assume the time is split into slots, denoted by $\mathbf{T} = \{t_1, t_2, ...\}$. Let n_t be the number of jobs submitted at time slot t. Denote $r_{ij,t}$ ($i \in \{1, ..., n_t\}, j \in \{1, ..., l\}$) as the amount of the type j resource allocated to job J_i at time slot t, ${r^u}_{ij,t}$ in $r_{ij,t}$, as the amount of unused type j resource allocated to job J_i at time slot t, $d_{ij,t}$ as job J_i's demand on type j resource at time slot t. Therefore, $r_{ij,t} = {r^u}_{ij,t} + d_{ij,t}$. For easy reference, Table 3.1 lists the main notations of CORP in this dissertation.

Table 3.1: Notations in CORP.

$\mathbf{J}$	A set of jobs
J_i	The ith job in $\mathbf{J}$
l	Number of resource types
$U_{a,t}$	Utilization of all resources at time t
N_p	Total number of PMs in the cloud system
N_v	Number of VMs in an area of the cloud system
θ	Significance level
η	Confidence level
C_{ij}	Capacity of v_i's type j resource
n_t	Number of jobs submitted at time t
${r^u}_{ij,t}$	Unused type j resource allocated to J_i at time t
$U_{j,t}$	System's utilization of type j resource at time t
$w_{j,t}$	Type j resource wastage ratio at time t
$w_{a,t}$	Overall resource wastage ratio at time t
$r_{ij,t}$	Amount of type j resource allocated to J_i at time t
$d_{ij,t}$	J_i's demand on type j resource at time t
h	Number of layers in deep neural network
N_n	Number of units per layer
$\hat{Y}_i$	Predicted J_i's unused resource using deep neural network
S	The set of states in HMM
H	Number of states in HMM
V	The set of possible observations
M	Number of observations
q_t	The state at time t
A	The state transition probability matrix
B	The observation probability matrix
π	The initial state distribution
$\mathcal{O}$	Observation sequence
$\mathcal{T}$	The length of observation sequence
$\gamma_t(i)$	Probability of being in state S_i at time t given the observation sequence $\mathcal{O}$ and the model λ
$\alpha_t(i)$	Forward variable
$\beta_t(i)$	Backward variable
$\hat{\sigma}$	Estimated SD for prediction errors
P_{th}	Probability threshold for prediction error

Hence, in the system, the resource utilization of type j resource at time slot t is

$$U_{j,t} = \frac{\sum_{i=1}^{n_t} d_{ij,t}}{\sum_{i=1}^{n_t} r_{ij,t}} \tag{3.1}$$

The overall resource utilization for all resources at time slot t is

$$U_{a,t} = \frac{\sum_{j=1}^{l}(\omega_j \sum_{i=1}^{n_t} d_{ij,t})}{\sum_{j=1}^{l}(\omega_j \sum_{i=1}^{n_t} r_{ij,t})} \tag{3.2}$$

where ω_j is the weight for type j resource, and $\sum_j^l \omega_j = 1$. The reason for setting different weights for different resource types is that sometimes some resources are more important than other resources. For example, CPU and MEM are more important than storage because storage is not the bottleneck resource [34], thus the weights of CPU and MEM are larger than storage. The type j resource wastage ratio at time slot t is

$$w_{j,t} = \frac{\sum_{i=1}^{n_t}(r_{ij,t} - d_{ij,t})}{\sum_{i=1}^{n_t} r_{ij,t}} \tag{3.3}$$

The overall resource wastage ratio for all resources at time slot t is

$$w_{a,t} = \frac{\sum_{j=1}^{l}(\omega_j \sum_{i=1}^{n_t}(r_{ij,t} - d_{ij,t}))}{\sum_{j=1}^{l}(\omega_j \sum_{i=1}^{n_t} r_{ij,t})} \tag{3.4}$$

Our objective is to minimize $w_{a,t}$, which will be shown in our formulated problem below.

3.2.1 Objective

Our problem of the VM resource allocation to jobs can be stated as below.

Problem Statement: Given a certain amount of resources (e.g., CPU, MEM, etc.), resource demands of each job, resource capacity constraints of VMs, how to allocate the VM resources to jobs to achieve high resource utilization while avoiding SLO violations as much as possible?

We denote $x_{ik} = \{0, 1\}$ as a binary variable representing if job J_i is assigned to VM v_k. In order to maximize resource utilization, we convert the problem of maximizing the resource utilization to minimizing the resource wastage. We present an integer linear programming (ILP) model [70] to minimize the resource wastage while satisfying the constraints of resource capacities of VMs.

$$Min\{w_{a,t}\} \tag{3.5}$$

$$s.t.\ r_{ij,t} \geq d_{ij,t}\ (\forall i \in \{1, ..., n_t\}, j \in \{1, ..., l\}) \tag{3.6}$$

$$\sum_{i=1}^{n_t} x_{ik} \cdot r_{ij,t} \leq R_{kj}\ (\forall i \in \{1, ..., n_t\}, j \in \{1, ..., l\}, k \in \{1, ..., N_v\}) \tag{3.7}$$

$$x_{ik} \in \{0, 1\}\ (\forall i \in \{1, ..., n_t\}, k \in \{1, ..., N_v\}) \tag{3.8}$$

$$d_{ij,t} \geq 0\ (\forall i \in \{1, ..., n_t\}, j \in \{1, ..., l\}) \tag{3.9}$$

where R_{kj} is the remaining type j resource of VM v_k. Constraint (3.6) is to ensure that the allocated resource meets each job's demand on each type of resource. Constraint (3.7) is to ensure that the amount of the allocated resource from each VM does not exceed its capacity of each resource type. Constraint (3.9) is to ensure that job J_i's resource demand on type j resource at time slot t is non-negative. We can use the CPLEX linear program solver [5] to solve this linear optimization problem.

The ILP optimization problem is an NP-hard problem and has high computational complexity [92]. Therefore, we propose a heuristic method called CORP. Specifically, CORP first accurately predicts the amount of temporally-unused allocated resource based on the historical data with the consideration of the non-existence of pattern and fluctuations of the amount of the unused resource. It then leverages complementarity of jobs' requirements on different resource types, and utilizes the packing strategy to allocate the unused resource to other jobs with a certain probability.

3.3 The Design of CORP

In Section 3.2, we describe the cooperative opportunistic resource provisioning problem and the objective of CORP. In this section, we present the design of CORP. In the following, Section 3.3.1 presents the prediction of the amount of temporally-unused resource and Section 3.3.2 presents the

unused resource allocation algorithm.

3.3.1 Prediction Process and Resource Preemption

In this dissertation, we use the deep learning together with HMM to accurately predict the temporally-unused resource with the consideration of the fluctuations of the amount of the unused resource, and then dynamically allocate the unused resource to users' jobs. We use L to denote the size of the window (the prediction horizon). After each time period L, we use the deep learning technique to make the predictions for the amount of the temporally-unused resource in a time window $\Delta W = (t, t + L]$, where t is the time when the prediction is made. After conducting the analysis of the Google trace from our system, we chose to make the predictions for a 1 minute window because short-lived jobs typically run minutes. We then use HMM to predict whether the amount of the temporally-unused resource will be in the peak or valley at $t + L$, based on which we adjust the predicted unused resource (e.g., CPU) by deep learning as the final predicted amount.

As shown in Figure 1.5, deep neural network utilizes multiple hidden layer structure for hierarchical feature learning. The multiple hidden layers enable the composition of features from lower layers, giving the potential of modeling complex data with fewer units. Compared with other machine learning methods, deep learning has the following inherent advantages. First, deep learning only needs the raw data for training without requiring sufficient high quality and truly representative past data [72]. Also, deep learning has better accuracy in many applications [112,134,136], so it can more accurately predict the real resource demands with the same given past data. More importantly, deep learning does not require that the historical data must have patterns, which is required by the methods like fast Fourier transform [34, 108].

The prediction process of the amount of the unused resource consists of three parts: 1) predicting the amount of unused resource using deep learning with HMM; 2) Prediction with confidence intervals; 3) Probabilistic-based resource preemption.

3.3.1.1 Predicting Unused Resource

3.3.1.1.1 Predicting Unused Resource Using Deep Learning

We use CPU as an example to illustrate the prediction of the amount of unused resource using deep learning. Each input data contains CPU utilization of a job at each slot in last $\triangle$ slots. To

build the deep neural network, for each input, there are three steps: feed-forward evaluation, back-propagation, weight update performed to update the model weights. In the feed-forward evaluation, the neurons take input data and perform simple operations on the data, and pass the results on to the up-layer neurons. In the back-propagation, the neurons calculate errors at output units and the errors are then back-propagated for each neuron in the lower layer of the output layer. In the weight update step, the errors are used to update the weights. Below, we introduce the details of each step.

Feed-forward evaluation: The output of each neuron i in layer d (called activation and denoted by $g_i(d)$) is computed as a function of its c inputs from neurons in the lower layer $d-1$. Let $w_{ij}(d-1,d)$ be the weight associated with a connection between neuron j in layer $d-1$ and neuron i in layer d, we have

$$g_i(d) = F((\Sigma_{j=1}^{c} w_{ij}(d-1,d) \cdot g_j(d-1)) + e_i) \tag{3.10}$$

where e_i is a bias term for the neuron. Equ. (3.10) is a sigmoid function, which is a nonlinear function associated with all neurons in the network, and it is more accurate [71].

Back-propagation: For each neuron i in the output layer, the error terms E are computed using the following equation:

$$E_i(d_h) = (t_i(d_h) - g_i(d_h)) \cdot F'(g_i(d_h)) \tag{3.11}$$

where $t(x)$ is the true value of the output and $F'(x)$ represents the derivative of $F(x)$. Next, these error terms are back-propagated for each neuron i in layer d connected to m neurons in layer $d+1$ summed as follows:

$$E_i(d) = (\sum_{j=1}^{m} E_j(d+1) \cdot w_{ji}(d,d+1)) \cdot F'(g_i(d)) \tag{3.12}$$

Weight updates: The error terms are used to update the weights and biases by using the following equation:

$$\Delta w_{ij}(d-1,d) = \mu \cdot E_i(d) \cdot g_j(d-1),\ \forall j = 1,...,c \tag{3.13}$$

where μ represents the learning rate parameter, and c is the number of inputs from neurons in layer $d-1$.

The process of these three steps is repeated for each input until the entire training dataset has been processed, which constitutes a training epoch. At the end of a training epoch, the model prediction error is computed as a held-out validation set. Basically, the training continues for

multiple training epochs, processing the training data set each time, until the validation set error converges to a low value. Finally, the deep neural network is built.

The deep learning algorithm for predicting the amount of unused resource is comprised of two parts: training and testing. For training, it first computes the hidden activation. Next, it computes the reconstructed output from the hidden activation. Then the algorithm computes the error gradient, and it back-propagates error gradient to update weight. For testing, the algorithm autoencodes the input and generates the output.

After the training, the deep neural network is built. To predict the unused resource of a job at time $t+L$, we input CPU utilization of a job at each slot in last $\triangle$ slots to the deep neural network, and the output is the amount of unused CPU resource of the job. The deep learning algorithm predicts the amount of unused resources of each job J_i in a time period, denoted by $\hat{Y}_i = (\hat{r}_{i1}, ..., \hat{r}_{il})$, where $\hat{r}_{ij}$ denotes the predicted amount of unused type j resource of job J_i. The resource usage of short-lived jobs sometimes fluctuates; it reaches a peak and a valley sometimes [108], which makes the actual amount of unused resource under fluctuations cannot be accurately predicted. To handle this problem, CORP then uses the HMM model to predict the peak and valley occurrences of the unused resource for prediction error correction. We present the details of the HMM model below.

Algorithm 2: Pseudocode for Predicting the Amount of Unused Resource Using Deep Learning

Input: VMs' unused resources at each slot within last Δ slots $R_{j,\xi} = (r_{j1,\xi}, ..., r_{jl,\xi})$, where $j \in \{1, ..., N\}$, $\xi \in \{t-\Delta, ..., t-1\}$.
Output: The unused resource of each VM $\hat{R}_j = (\hat{r}_{j1}, ..., \hat{r}_{jl})$, $j \in \{1, ..., N\}$.

```
Fix a number of m hidden units, and a number of h of hidden layers.
for i ← 1 to N_v do
    for each J_j in v_i do //Training
        for each training iteration do
            for each hidden layer do
                Compute activation g_u(d) from input g_v(d − 1)
            Compute error gradient //Equs. (3.11), (3.12)
            Back-propagate error gradient to update weight
              parameters //Equ. (3.13)
    for each J_j in v_i do //Testing
        Autoencode g_i and generate ĝ_i
    Calculate v_i's unused resource (R̂_j) by subtracting the allocated resource from the
    capacity of the VM and adding the unused resource of all jobs in v_i.
Return (R̂_1, ..., R̂_N)
```

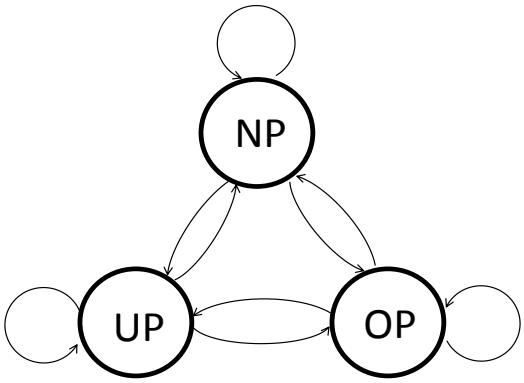


Figure 3.1: Hidden Markov Model with three states: over-provisioning (OP), normal-provisioning (NP), and under-provisioning (UP).

3.3.1.1.2 Predicting Fluctuations of Unused Resource Using HMM

We use the HMM model to predict the peak and valley occurrences of unused resource. We use CPU as an example to illustrate the process, and the method can be directly applied to other resource types. In the HMM model, we define three observation symbols: peak, center and valley. Peak refers to a sharp increase of the amount of unused resource; center refers to no change or a moderate change of the amount of unused resource; valley refers to a sharp decrease of the amount of unused resource. Given a set of historical data, let max_{cpu}, m_{cpu} and min_{cpu} be the maximum amount, average amount and minimum amount of unused CPU resource in the historical data, respectively. We split the interval $[min_{cpu}, max_{cpu}]$ into 3 subintervals: $[min_{cpu}, min_{cpu} + \frac{1}{2}(m_{cpu} - min_{cpu})]$, $(min_{cpu} + \frac{1}{2}(m_{cpu} - min_{cpu}), m_{cpu} + \frac{1}{2}(max_{cpu} - m_{cpu}))$, $[m_{cpu} + \frac{1}{2}(max_{cpu} - m_{cpu}), max_{cpu}]$. We call these three parts as peak, center, valley, respectively, which are used to categorize the observation symbols of the HMM model. The corresponding (hidden) states that determine the observation symbols are over-provisioning (OP), normal-provisioning (NP), under-provisioning (UP), respectively (see Figure 3.1).

Denote $S = \{S_1, ..., S_H\}$ $(H = 3)$ as the set of states, q_t as the state at t, and $Q = q_1 q_2 ... q_T$ as a state sequence. Let $V = \{1, ..., M\}$ $(M = 3)$ be the set of possible observation symbols per state, and $O = \{O_1, ..., O_T\}$ $(O_i \in V,\ \forall i = 1, ..., T)$ be the observation sequence, where M is the number of observation symbols[1] $(1, 2, 3$ represent "peak", "center" and "valley" regions, respectively) and T is the length of observation sequence. To determine the observation symbols, we consider the time

[1]The number of states H does not necessary equal the possible observation symbols per state M, and the HMM model in our work is a special case.

interval between two consecutive observation time slots j and $j+1$ ($j = 1, ..., T-1$) as a window, and we divide the window into $L-1$ subwindows. Let Δ_j be the difference between the maximum amount of unused resource and the minimum amount of unused resource in the window. If Δ_j falls in $[min_{cpu}, min_{cpu} + \frac{1}{2}(m_{cpu} - min_{cpu})]$, then we consider the observation symbol at $j+1$ is valley; if Δ_j falls in $(min_{cpu} + \frac{1}{2}(m_{cpu} - min_{cpu}), m_{cpu} + \frac{1}{2}(max_{cpu} - m_{cpu}))$, then we consider the observation symbol at $j+1$ is center; otherwise, we consider the observation symbol at $j+1$ is peak. Then, the state transition probability matrix is

$$A = \{a_{ij}\} \ (a_{ij} = P\{q_{t+1} = S_j | q_t = S_i\}, \ 1 \leq i, j \leq H) \tag{3.14}$$

where the state transition coefficients have the following properties

$$a_{ij} \geq 0 \tag{3.15a}$$

$$\sum_{j=1}^{H} a_{ij} = 1 \tag{3.15b}$$

The observation probability matrix $B = \{b_j(k)\}$ is

$$B = \{b_j(k)\} \ (b_j(k) = P\{O_t = k | q_t = S_j\}, 1 \leq j \leq H, 1 \leq k \leq M) \tag{3.16}$$

where $b_j(k)$ is the probability that the observation symbol is k given the sate at t is S_j. Thus, the initial state distribution is

$$\pi = \{\pi_i\} \ (\pi_i = P\{q_1 = S_i\}, \ 1 \leq i \leq H) \tag{3.17}$$

Given the model $\lambda = (A, B, \pi)$ and an observation sequence O, our goal is to find the most likely state sequence. Specifically, we aim to maximize the expected number of correct states for the HMM. We define $\gamma_t(i)$ as the probability of being in state S_i at time t, given the observation sequence O and the model λ:

$$\gamma_t(i) = P\{q_t = S_i | O, \lambda\} \tag{3.18}$$

Equ. (3.18) can be simplified with the forward-backward variables as follows:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} \tag{3.19}$$

where $\alpha_t(i)$ is the forward variable defined as

$$\alpha_t(i) = P(O_1 O_2 \cdots O_t, q_t = S_i|\lambda) \tag{3.20}$$

$\alpha_t(i)$ can be solved inductively using Algorithm 3 [94].

Algorithm 3: Pseudocode for solving forward variable $\alpha_t(i)$ inductively

```
for i ← 1 to H do
    α_1(i) = π_i b_i(O_1)  //Initialization
for t ← 1 to T − 1 do //Induction
    for j ← 1 to H do
        α_{t+1}(j) = [Σ_{i=1}^{H} α_t(i) a_ij] b_j(O_{t+1})
P(O|λ) = Σ_{i=1}^{H} α_T(i)  //Termination
```

$\beta_t(i)$ is the backward variable defined as

$$\beta_t(i) = P(O_{t+1} O_{t+2} \cdots O_T|q_t = S_i, \lambda) \tag{3.21}$$

$\beta_t(i)$ can be solved inductively using Algorithm 4 [94].

Algorithm 4: Pseudocode for solving backward variable $\beta_t(i)$ inductively

```
for i ← 1 to H do
    β_T(i) = 1  //Initialization
for t ← T − 1 to 1 do
    for i ← 1 to H do //Induction
        β_t(i) = [Σ_{j=1}^{H} a_ij] b_j(O_{t+1}) β_{t+1}(j)
```

By using $\gamma_t(i)$, we can solve for the individually most likely state q_t at time t, as

$$q_t = argmax_{0 \le i \le M-1}[\gamma_t(i)],\ 1 \le t \le T \tag{3.22}$$

Equ. (3.22) chooses the most likely state for each t to maximize the expected number of correct states. In implementation, we use Viterbi algorithm to find the single best state sequence (path), denoted

by Q^*=$q_1^* q_2^* ... q_L^*$, i.e., maximizing $P(Q, O|\lambda)$ which is equivalent to maximizing $P(Q|O, \lambda)$ [94], and we use the method in [113] to re-estimate the parameters A, B, π in the model.

Based on the work [54], the probability distribution of the next fluctuation observation of the amount of unused resource can be estimated as

$$E_{P_{T+1}(k)} = \sum_{j=1}^{H} P(q_{T+1} = S_j | q_T = q_L^*) \cdot b_j(k) \ (k \in \{1, ..., M\}) \tag{3.23}$$

We consider the observation symbol which has the highest value of $E_{P_{T+1}(k)}$ as the observation symbol of the next time $T+1$, that is, $k|_{E_{P_{T+1}(k)} = max_{u=1}^{M}(E_{P_{T+1}(u)})}$.

Given the resource utilization of jobs, CORP uses HMM to predict the fluctuations of the amount of unused resource (i.e., peak, center, valley symbols) for the next time period. Recall that we use deep learning to perform predictions for the amount of temporally-unused resource at the end of each window L, denoted by $\hat{Y}_j$. Then, we use HMM to predict the fluctuations of the amount of unused resource and adjusts the predicted amount accordingly. We use $\hat{u}_{t+L}$ to represent the predicted unused resource with prediction error correction at time t for a future time $t+L$. Specifically, if the predicted observation symbol of unused CPU resource falls in the valley, CORP reduces the predicted amount by $\hat{u}_{t+L} = \hat{r}_{j1} - min(h_{cpu} - m_{cpu}, m_{cpu} - l_{cpu})$ (suppose the first resource type in $\hat{Y}_j$ is CPU), where m_{cpu} is the average value of unused CPU resource in the historical data, h_{cpu} is the highest amount of unused resource within a period, and l_{cpu} is the lowest amount of unused resource within a period. If the predicted unused CPU resource falls in the peak, CORP makes the adjustment by $\hat{u}_{t+L} = \hat{r}_{j1} + min(h_{cpu} - m_{cpu}, m_{cpu} - l_{cpu})$. The reasons for using $min(h_{cpu} - m_{cpu}, m_{cpu} - l_{cpu})$ to correct overestimation (or underestimation) errors are as follows. First, $h_{cpu} - m_{cpu}$ and $m_{cpu} - l_{cpu}$ indicate the deviation between the amount of unused resource in peak and the average of the unused resource, and the amount of unused resource in valley and the average of unused resource. The predicted amount may be close to m_{cpu}. Therefore, such adjustment can make the predicted unused resource closer to the actual amount of unused resource if it is in the peak or valley. Second, we use min because it is more conservative for ensuring sufficient resource being able to allocated to jobs.

3.3.1.2 Predicting Confidence Intervals

To ensure the accuracy of the prediction, we use a confidence interval for the probability that the resource will be available. The confidence interval is an estimate of the range of values within which the true value should lie with a certain confidence level (in the form of probability denoted by η). The higher the confidence level, the wider the confidence interval, and the more conservative the predictions. The confidence interval calculation depends on the variance of the prediction errors and the confidence level η. Let $\theta = 1 - \eta$ be the significance level. The confidence interval is

$$[\hat{u}_{t+L} - \hat{\sigma} \cdot z_{\frac{\theta}{2}}, \hat{u}_{t+L} + \hat{\sigma} \cdot z_{\frac{\theta}{2}}] \tag{3.24}$$

where $\hat{u}_{t+L}$ is the forecast for unused resource at time t for a future time $t + L$, $\hat{\sigma}$ is the estimated standard deviation (SD) for the prediction errors, and $z_{\frac{\theta}{2}}$ is the value for the $100 \cdot \frac{\theta}{2}$ percentile in the standard normal distribution.

In order to provide SLOs to users, predictions for a certain amount of time period is required so that users' jobs can complete. We generated the predictions for a time window $\Delta_W = (t, t + L]$, where t is the time when the prediction is made, and L is the size of the window (the prediction horizon), corresponding to the length of the observation sequence T.

After conducting the analysis of the trace from our system, we chose to make the predictions for a 1 minute window. We performed new predictions at the end of each window. Each one is a prediction cycle.

Based on a given confidence interval, the predicted amount of unused resource for time $t+L$ is adjusted by the following equation

$$\hat{u}_{t+L} = \hat{u}_{t+L} - \hat{\sigma} \cdot z_{\frac{\theta}{2}} \tag{3.25}$$

We use the lower bound of the confidence interval in Equ. (3.25) because the underestimation of the unused resource makes it conservative in reallocating allocated resources, thus avoiding SLO violations.

Based on the historical data with prediction error samples, we calculate the prediction error in a time window as follows

$$\delta_{t+\tau} = u_{t+\tau} - \hat{u}_{t+L}, \forall \tau \in [1, L] \tag{3.26}$$

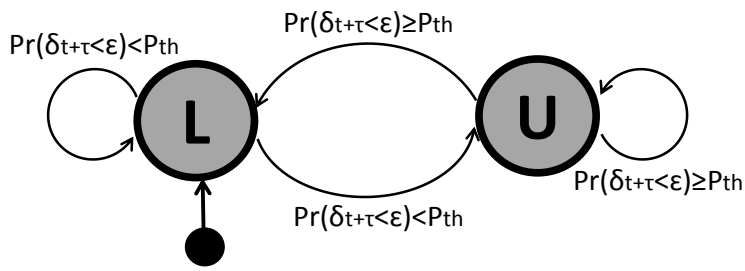


Figure 3.2: Probabilistic-based Resource Preemption with FSM: L represents Locked state, U represents Unlocked state.

That is, we calculate the prediction error for each time slot in the window $\tau \in [1, L]$ by subtracting the predicted unused resource at time t from the actual amount of unused resource at each time slot.

3.3.1.3 Probabilistic-based Resource Preemption

We use two states to represent the status of the predicted temporarily-unused resource: locked and unlocked. One unlocked state indicates that the unused resource can be preempted, and the locked state indicates that the unused resource cannot be preempted. Let ε denote pre-specified prediction error tolerance and P_{th} denote a pre-defined probability threshold. For a predicted temporarily-unused resource with prediction error δ_{t+L}, if δ_{t+L} satisfies

$$Pr(0 \leq \delta_{t+L} < \varepsilon) \geq P_{th} \tag{3.27}$$

then it can be allocated to a new arriving job and its state is "unlocked", otherwise, its state is "locked".

Figure 3.2 illustrates the probabilistic-based resource preemption using finite state machine (FSM). The locked state is the start state, and the probability of the prediction error is the input. If Equ. (3.27) is satisfied, then the unused resource can be allocated to other jobs. Otherwise, the unused resource should be kept for its allocated job. This is because the prediction accuracy is not high enough and the actual amount of unused resource may not meet the resource requirements of other jobs. When the state goes to "unlocked", the unused resource is allowed to be preempted. After the current state goes to "unlocked", if the unused resource can be preempted, then the state keeps being "unlocked". Otherwise the state goes back to "locked", and the unused resource cannot be preempted.

After predicting the amount of unused resource, CORP can determine if the unused resource

can be allocated to jobs. CORP uses a resource packing strategy by leveraging complementarity of jobs' requirements on different resource types (e.g., CPU, MEM, Storage, etc.) and allocates resource to jobs (packing jobs to VMs), which can reduce the resource fragmentation and further increase the resource utilization. The details are illustrated in Section 3.3.2.

Algorithm 5: Pseudocode for Finding Lowest Remaining Volume VM (LRV) Algorithm

Input: The predicted unused resource types of VMs $\hat{R}_j = (\hat{r}_{j1}, ..., \hat{r}_{jl})$, the resource demands of a job or a job packed with another job, the area $\mathcal{A}_k$ where the job(s) submitted.

Output: The VM with unused resource closest to the demands of job(s) $D_i = (d_{i1}, ..., d_{il})$.

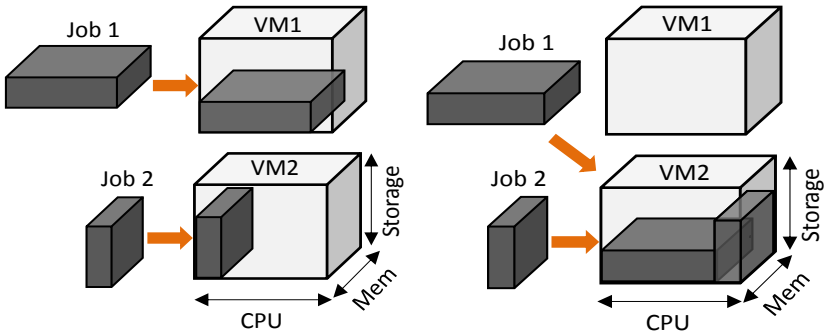
```
1 V_min ← E; min ← 0; w ← 0 //E is a large positive real
2 for each v_j in A_k do
3     if R̂_j ⪰ D_i && Pr(0 ≤ δ_{t+L} < ε) ≥ P_th then //∀u ∈ {1,...,l}, r̂_ju ≥ d_iu
4         w ← w + 1
5         Compute V(j) //Remaining volume of VM v_j
6         if V(j) < V_min then
7             min ← j
8 Return min
```

3.3.2 Resource Allocation Algorithm

CORP periodically predicts the allocated and unused resources in each VM. For newly arriving jobs, CORP conducts packing to pack complementary jobs, and then allocates unlocked or unallocated resources to the packed jobs based on their resource demands. The job packing is used to avoid resource fragmentation and achieve high resource utilization. In the following, we first present an example to show the complementary job packing, and then explain its algorithm. Then, we present the resource allocation algorithm that reallocates unlocked predicted unused resources to newly arriving jobs. Finally, we present an example to show this algorithm.

Figure 3.3 shows an example illustrating how packing strategy decreases the resource fragmentation and increases resource utilization. In Figure 3.3(a), job 1 (CPU intensive) and job 2 (storage intensive) are assigned to VM1 and VM2, respectively, which increases the resource fragmentation of VMs. However, in Figure 3.3(b), job 1 and job 2 are packed first and then assigned to VM2, which releases VM1, and thus decreases the resource fragmentation of VMs and increases the resource utilization.

Each job has a dominant resource, defined as the one that requires the most amount of re-



(a) Without packing (b) With packing

Figure 3.3: Allocate the resource of VMs to the jobs w/o and w/ packing strategy.

source. CORP first packs the jobs with complementary dominant resources such that the summation of the deviation of the two jobs' resource demands on each resource type is the largest. Given a list of jobs, CORP fetches each job J_i, and tries to find its complementary job from the list to pack with J_i. Note that it is possible that job J_i's complementary job cannot be found from the list. In this case, the job J_i solely constitutes an entity to be allocated with resources in a VM. To find J_i's complementary job, CORP calculates its deviation with every other job J_j if J_j has different dominant resource from J_i. The deviation is calculated by $DV(j,i) = \sum_{k=1}^{l}((d_{jk} - \frac{d_{jk}+d_{ik}}{2})^2 + (d_{ik} - \frac{d_{jk}+d_{ik}}{2})^2)$. Finally, the job with the highest deviation value is the complementary job of J_i.

After the job packing, CORP needs to assign each job entity (packed jobs or a job) to a VM with unlocked predicted unused resources. Among the VMs with unlocked predicted unused resources that can satisfy the resource demand of the job entity, we will choose the VM that has the least remaining resources (called most matched VM) in order to more fully utilize resources. If predicted unused resources cannot satisfy the resource demand of the job entity, unallocated resources in a VM will be used for the job entity using the same method. To find the most matched VM, we introduce a concept called unused resource volume. Suppose the vector of the maximum capacity of each resource type among all VMs is $\mathbf{C}' =< C'_1, C'_2, \ldots, C'_l >$. Assume that the amount of predicted unused resource of VM j is $\hat{R}_j = (\hat{r}_{j1}, ..., \hat{r}_{jl})$. Then, the unused resource volume of VM j is calculated by

$$volume_j = \sum_{k=1}^{l} \hat{r}_{jk}/C'_k \tag{3.28}$$

The VM satisfying the resource demand and has the smallest unallocated volume is the most matched VM.

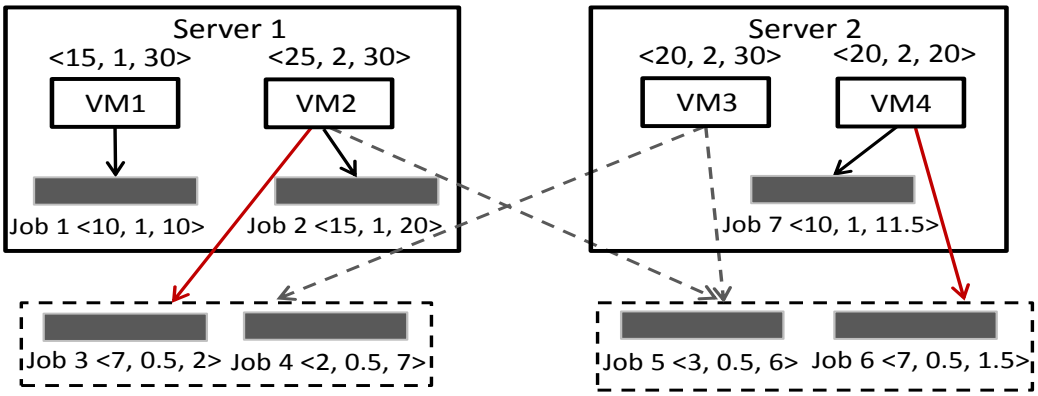


Figure 3.4: Allocate unused resource to (packed) jobs with low resource wastage.

Figure 3.4 shows an example illustrating the process of job packing and how CORP allocates the predicted unused resource to a job entity. For VMs, the numerical values indicate the capacities of different resource types. For jobs, the numerical values indicate the resource demands of jobs. Job 3, job 4, job 5 and job 6 are new arriving jobs. The dominant resource of jobs 3 and 6 is CPU, and the dominant resource of jobs 4 and 5 is storage. CORP first conducts job packing. The resource demand deviation of job 3 and job 4 is 25, and that of job 3 and job 5 is 16. Since $25 > 16$, job 3 and job 4 are packed together. Similarly, job 5 and job 6 are packed together. We denote the job entities as (job 3, job 4) and (job 5 and job 6). The maximum CPU, MEM and storage of all VMs among both servers are $\mathbf{C}' =< 25, 2, 30 >$. If the amount of unlocked predicted unused resource of VMs 1-4 are as follows: $< 5, 0, 20 >$, $< 10, 1, 10 >$, $< 20, 2, 30 >$ and $< 10, 1, 8.5 >$, respectively, based on Equ. (3.28), their unused resource volumes are 0.867, 1.233, 2.8, 1.183, respectively. To allocate resources to entity (job 3, job 4), CORP first checks if the VMs' predicted unused resources can satisfy the demands on each type of resource of the entity. Then, CORP chooses the VM that has the smallest unused resource volume to be allocated to the job entity. In this example, VM1 and VM4 cannot satisfy its resource requirements of the packed job (jobs 3, 4). By comparing the unused resource volumes of VM2 and VM3, because $1.233 < 2.8$, then CORP chooses VM2 rather than VM3 and allocates its temporarily-unused resource to the packed job (job 3 and job 4). Similarly, the predicted unused resource of VM1 cannot satisfy the resource requirements of the packed job (job 5, job 6). By comparing the unused resource volumes of VM2, VM3 and VM4, because $1.183 < 1.233 < 2.8$, then CORP chooses VM4 and allocates its temporarily-unused resource to the packed job (job 5 and job 6). The above process of allocating unused resource to jobs also

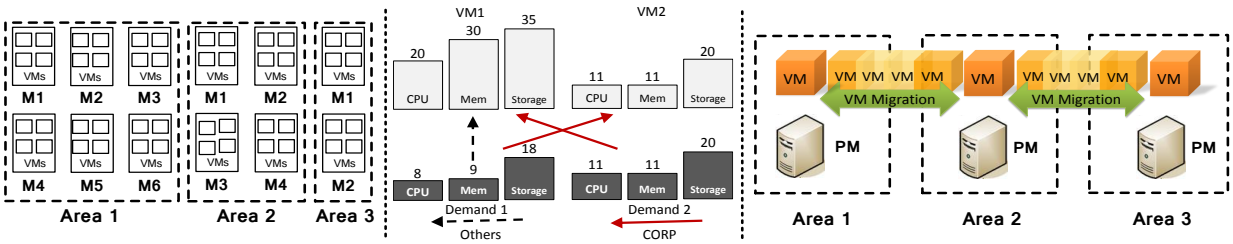


Figure 3.5: The architecture of CORP system for resource provisioning.

applies to the single job case.

Figure 3.5 shows the design of CORP system for resource provisioning. We deploy the servers (PMs) in different areas based on the intensity of jobs for processing. In each server, the server allocates resources to VMs. In each area, users submit their jobs to the system, and CORP allocates the resources to the jobs based on the predicted amount of unused resources. For a

Algorithm 6: Pseudocode for Job Packing (JP) Algorithm

Input: A job J_i's resource demand, the area $\mathcal{A}_k$ where J_i submitted
Output: A cooperative job J_j packed with J_i.

$u \leftarrow 0$
for *each* J_j $(J_j \in \boldsymbol{J})$ *in* $\mathcal{A}_k$ **do**
  if J_j *and* J_i *have different dominate resource* **then**
    $DV(j,i) = \sum_{K=1}^{l}((d_{jK} - \frac{d_{jK}+d_{iK}}{2})^2 + (d_{iK} - \frac{d_{jK}+d_{iK}}{2})^2)$
    $u \leftarrow u + 1$
    $B(u).elem \leftarrow DV(j,i)$
    $B(u).index \leftarrow j$
Sort $B(i)$, $\forall i \in \{1, ..., u\}$ `//Descending order based on elem`
for *each* $B(i) \in B$ **do** `//Travel from beginning to end`
  if $J_{B(i).index} \in \boldsymbol{J}$ *&&* $J_{B(i).index}$ *is in* $\mathcal{A}_k$ **then**
    Return $B(i).index$
  else
    Return $NULL$

new arriving job (or packed job), CORP first checks the servers located in the area which has the smallest geographic distance to the place where the job is submitted. Then CORP checks the unused resource of the VMs allocated to other jobs to see if there exists some VM that can satisfy the job's demands on all types of resources. If there exist some VMs allocated to other jobs can satisfy the job's demands on all resources types, CORP chooses the VMs with the probability of the prediction error being less than ε and greater than or equal to 0 is no less than a threshold P_{th} from these VMs and lists them as candidate VMs for running the job. Then CORP chooses a VM from the

candidate VMs and allocates the unused resource to the job. If no VM allocated to other job(s) can satisfy both the job's demands on resources and the condition that the probability of the prediction error being less than ε and greater than or equal to 0 is no less than P_{th}, then CORP chooses an unallocated VM satisfying both these two conditions and allocate it to the job. In order to avoid resource fragmentation and achieve high resource utilization, CORP uses the following method which can be divided into two cases, to allocate the resource to a new arriving job.

Algorithm 7: Pseudocode for Cooperative Opportunistic Resource Provisioning

Input: n jobs' ($\mathbf{J} = \{J_1, ..., J_n\}$) resource demands $D_i = (d_{i1}, ..., d_{il})$ ($i \in \{1, ..., n\}$), VM's unused resources at each slot within last Δ slots

1 Predict unused resource $\hat{R}_j$ ($j = \{1, ..., N_v\}$) of each VM using Algorithm 2 //$\hat{R}_j = (\hat{r}_{j1}, ..., \hat{r}_{jl})$
2 Predict the fluctuations of unused resource using HMM
3 **if** *(The observation symbol of the predicted amount of the unused resource is peak or valley)* **then**
4 Correct the predicted unused resource
5 Group jobs in the same area $\mathcal{A}_i$ together //$\mathcal{A}_i = \{\mathcal{A}_1, ...\mathcal{A}_\Lambda\}$
6 **for** *each* $\mathcal{A}_i \in \mathcal{A}$ **do**
7 **for** *each* J_j ($J_j \in \boldsymbol{J}$) *in* $\mathcal{A}_i$ **do**
8 CPT(J_j)$\leftarrow$ Call JP (J_i, $\mathcal{A}_i$) `//A job packed with` J_i
9 $min \leftarrow 0$
10 **if** *CPT(J_j)$\neq$ NULL* **then** `//Find a cooperative job packed with` J_i
11 $D_i \leftarrow D_i + D_{CPT(J_i)}$
12 $min \leftarrow Call\ LRV\ (D_i, \{J_i, J_{B(i).index}\})$
13 Allocate v_{min}'s unused resource to $\{J_i, J_{B(i).index}\}$
14 $\mathbf{J} \leftarrow \mathbf{J} \setminus \{J_i, J_{B(i).index}\}$
15 **else**
16 $min \leftarrow Call\ LRV\ (D_i, J_i)$
17 Allocate v_{min}'s unused resource to J_i
18 $\mathbf{J} \leftarrow \mathbf{J} \setminus \{J_i\}$

Case 1: Reallocate a VM's temporarily-unused resource to a new arriving job.

Case 2: Allocate the resource of an unallocated VM to new arriving job(s).

Algorithm 7 shows the pseudocode of the CORP algorithm for resource provisioning. The algorithm first predicts the unused resource with historical data based on Algorithm 2 (Line 1). It then predict how long the unused resource will not be used (Line 2). Then it checks if the unused resources satisfy the job's demands. For those VMs whose remaining unused resources can satisfy the job's demands and the probability that the available time of the unused resource is greater than a threshold, CORP chooses a VM whose remaining resource is closest to the demands of the job.

Table 3.2: Parameter settings.

Parameter	Meaning	Setting
N_p	# of servers	30-50
N_v	# of VMs	100-400
$\|\mathbf{J}\|$	# of jobs	50-300
l	# of resource types	3
h	# of layers in deep neural network	4 [80]
N_n	# of units per layer	50
H	# of states in HMM	3
P_{th}	A Prob. threshold	0.95
θ	Significance level	5%-30%
η	Confidence level	50%-90%

3.4 Performance Evaluation

In this section, we present our trace-driven experimental results on a large-scale real cluster, Clemson University's high-performance computing (HPC) resource [9], and Amazon EC2 [2], respectively. To show the performance of CORP, we compared CORP with RCCR [34], CloudScale [108], DRA [103] in various scenarios since all these methods share the same objective of maximizing the resource utilization while avoiding SLO violation.

RCCR uses time series forecasting to predict the fraction of unused resources that will almost certainly not be required in the future based on historical resource usage patterns and allocates the unused resource to long-term service jobs in an opportunistic manner. CloudScale employs online resource demand prediction and prediction error handling to adaptively allocate the resources on PMs to VMs to achieve high resource utilization. Unlike RCCR, CORP first predicts the amount of allocated but unused resource using the deep learning technique, in which the accuracy does not rely on the existence of resource utilization patterns of short-lived jobs. To well meet the requirement of time-varying job resource demands, CORP additionally considers the fluctuations of unused resource and uses HMM model to correct prediction errors. Also, CORP leverages complementarity of jobs' requirements on different resource types and packs jobs with complementary resource requirements to VMs to reduce the resource fragmentation and further increase the resource utilization. DRA provides the cloud customer with the abstraction of buying bulk capacity (rather than pre-defined VM configurations based on the peak demands of the applications). DRA first purchases capacity for the customers, and then re-distributes the purchased capacity among customer's VMs based on their demand. Specifically, DRA considers the share value and the demand value of VMs and allocates the aggregate amount of capacity purchased by the customers among the VMs in an equitable manner

taking into account shares and not giving the VMs more than what they demand. DRA presents two distributed resource allocation algorithms that periodically re-distribute the purchased capacity among the tenant's (customer's) VMs based on their dynamic demand.

In the implementation on the real cluster, we applied for 50 nodes, and we simulated a node as a PM, and we simulated a logic disk as a VM; in the implementation on Amazon EC2, we applied for 30 nodes, each node is simulated as a VM. For CORP, we first used the deep learning algorithm to predict the amount of unused resource of jobs running on the VMs based on the historical resource usage data from the Google trace. Next, we used the HMM model to predict fluctuations of the amount of unused resource of jobs, and we adjusted the predicted amount for the peak and valley of the unused resource. Then, we packed two jobs with complementary dominant resources such that the summation of the deviation of the two jobs' resource demands on each resource type is the largest (see Section 3.3.2). Finally, we chose the VM that has the least remaining resources that can satisfy the resource demands of job(s) and allocated it to the job(s) (see Section 3.3.2) (We know the capacity for each type of resource of a VM, and we can know the amount of unused resources of each VM after we getting the amount of unused resource of jobs and amount of resource allocated to jobs.). For RCCR, we first used a time series forecasting technique, i.e., Exponential Smoothing (ETS), to predict the amount of unused resource of VMs. Then we calculated confidence intervals and chose the lower bound of the confidence interval as the predicted value for a time window ΔW. Finally, we randomly chose a VM that can satisfy the resource demands of a job and allocated resource to the job without considering job packing. For CloudScale, we first used the prediction model developed in [56] and a discrete-time Markov chain to predict the amount of unused resource of VMs based on historical resource usage data. Then we extracted the burst pattern to get the padding value and calculated the prediction errors by subtracting the predicted amount of unused resource from the actual amount of unused resource. Next, we used the adaptive padding that is based on the recent burstiness of resource usage and recent prediction errors to correct the prediction errors. Finally, we also randomly chose a VM that can satisfy the resource demands of the job and allocated the unallocated resource to the job without considering job packing. For DRA, we simulated the purchased capacity as the total amount of resource of all VMs that are used to be allocated to jobs. For each VM, we defined two properties: share and demand. We statically set the share value at the time of VM creation so that the VMs had a mix of high, medium and low shares that correspond to a ratio of 4:2:1, respectively. We used the run-time software to periodically

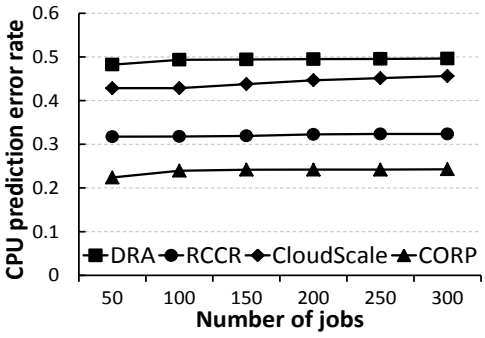


Figure 3.6: Prediction error rate vs. # of jobs across different methods on a real cluster.

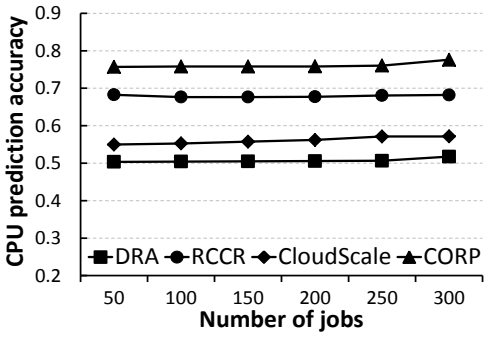


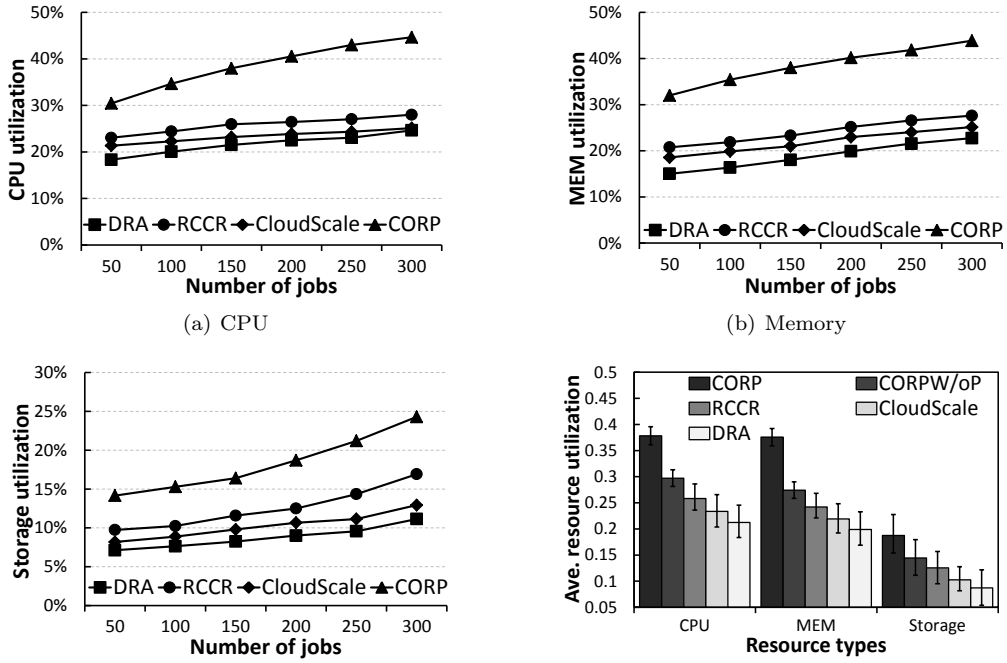
Figure 3.7: Prediction accuracy vs. # of jobs across different methods on a real cluster.

estimate the amount of unused resource of VMs based on the historical resource usage data. Then, we redistributed the purchased capacity among different VMs based on their shares and demands. Finally, we randomly chose a VM that can satisfy the resource demands of the job and allocated the unallocated resource to the job.

We first deployed our testbed on the real cluster using 50 servers and then conducted experiments on the real-world Amazon EC2 using 30 servers. The servers in the real cluster are from HP SL230 servers (E5-2665 CPU, 64GB memory) [9] The servers in Amazon EC2 are from commercial product HP ProLiant ML110 G5 servers (2660 MIPS CPU, 4GB memory) [37]. In both experiments, each server is set to have 1GB/s bandwidth and 720GB disk storage capacity. In both experiments, we used the trace from Google [7] which records the resource requirements and usage of tasks every 5 minutes, and we transformed the 5-minute trace into 10-second trace and we set the CPU, memory and storage consumption for each job based on the Google trace [7]. In the trace, we considered the tasks of jobs in the trace as short-lived jobs, the bandwidth consumption for each short-lived job is set as 0.02 MB/s [110]. SLO is specified by using a threshold on the response time of a job, and the threshold is set based on the execution time of a task in the trace. To fully verify the performances of our method and the other three methods, we varied the number of jobs from 50 to 300 with step size of 50. Table 4.2 shows the parameter settings in our experiment unless otherwise specified.

3.4.1 Experimental Results on the Real Cluster

We first calculated the prediction error of CPU by subtracting the predicted amount of

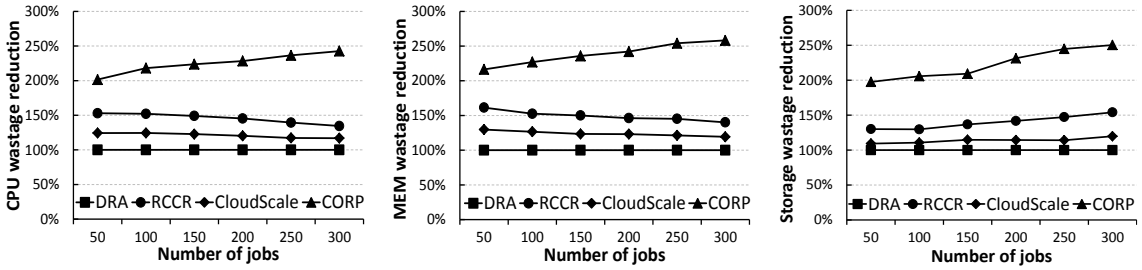


(a) CPU (b) Memory

(c) Storage (d) Average

Figure 3.8: Utilizations of different resource types vs. number of jobs of different methods on a real cluster.

unused resource from the actual amount of unused resource for each job. Then we calculated the ratio of the correctly predicted jobs (the jobs whose prediction errors are within $[0, \varepsilon)$) to the number of jobs as the prediction error rate which ranges from 0 to 1. Figure 3.6 shows the relationship between the prediction error rate and the number of jobs. We see that the prediction error rate follows CORP<RCCR<CloudScale<DRA. The prediction error rate in RCCR is higher than that in CORP because CORP takes advantage of deep learning which can detect complex interactions among features and can learn low-level features from minimally processed raw data. Also, the prediction accuracy of the deep learning algorithm does not rely on the assumption that the historical data for prediction has patterns, which can decrease the prediction error rate generated by the data pattern assumption, and it is suitable for short-lived jobs. Moreover, CORP adequately considers the fluctuations of the unused resource caused by the bursts of jobs' resource demands and utilizes HMM model to correct the prediction errors, which reduces the prediction error rate. However, RCCR uses a time series forecasting method of which the accuracy relies on the existence of patterns in resource usage [36, 115], to predict the unused resource for long-running service jobs, which can increase the error rate when the resource usage does not have patterns. Also, RCCR does not



(a) CPU (b) Memory (c) Storage

Figure 3.9: Resource wastage reduction vs. number of jobs of different methods on a real cluster.

adequately consider the fluctuations of the unused resource of short-lived jobs caused by their bursts of resource demands, which can increase the error rate, and thus is not suitable for short-lived jobs. CloudScale generates higher prediction error rate than CORP and RCCR because CloudScale's prediction accuracy relies on the assumption of the existence of data patterns in the historical data, which can increase the error rate caused by the data pattern assumption. Although CloudScale uses a multi-step Markov prediction to dealing with the prediction when pattern is not found, it has limited prediction accuracy since the correlation between the resource prediction model and the actual resource demand becomes weaker. Also, CloudScale does not utilize confidence levels to make appropriately-conservative predictions and thus reduce the error rate. The prediction error rate in DRA is higher than all the other methods because DRA does not consider the fluctuations of unused resource caused by the bursts of jobs' resource demands, which can increase the prediction error rate. Also, DRA uses the run-time software to estimate the resource periodically, the accuracy of which also relies on the existence of patterns in the training data. In addition, DRA does not utilize confidence levels to make appropriately-conservative predictions and reduce the error rate.

Figure 3.7 shows the relationship between the prediction accuracy and the number of jobs. We see that the prediction accuracy follows CORP>RCCR>CloudScale>DRA, which is consistent with the result in Figure 3.6. The prediction accuracy in CORP is higher than that in RCCR because CORP utilizes deep learning to predict the unused resource and increase the prediction accuracy. Also, CORP considers the fluctuations of the unused resource caused by the bursts of jobs' resource demands and uses the HMM model to correct the prediction errors, which increases the prediction accuracy. CloudScale generates lower prediction accuracy than CORP and RCCR

because CloudScale relies on the assumption of the existence of data patterns in the historical data,

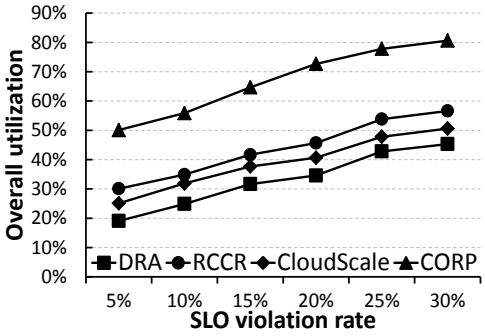


Figure 3.10: Resource utilization vs. SLO violation rate on a real cluster.

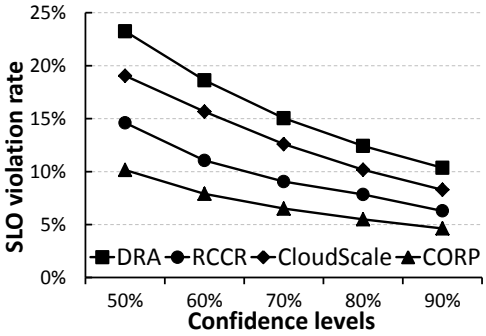


Figure 3.11: SLO violation rate vs. confidence levels on a real cluster.

which can increase the error rate caused by the data pattern assumption and thus decrease the prediction accuracy. The prediction accuracy in DRA is lower than all the other methods because DRA neglects the fluctuations of unused resource caused by the bursts of jobs' resource demands, which can decrease the prediction accuracy by increasing the prediction error rate.

We used Equ. (3.1) to calculate the resource utilization of type j resource. Figure 3.8 shows the relationship between the resource utilization and the number of jobs. We observe that the resource utilization follows CORP>RCCR>CloudScale>DRA. The resource utilization in CORP is higher than that in RCCR because CORP leverages complementarity of jobs' demands on different resource types and uses a job packing strategy to reduce the resource fragmentation. Also, CORP uses deep learning to predict the unused resource, and adequately considers the fluctuations of short-lived jobs' unused resource, and uses the HMM model to correct the prediction error, and then dynamically allocates the resource to jobs to well meet the requirement of time-varying resource demands and decreases the probability of resource over-provisioning, which is suitable for short-lived jobs. However, RCCR uses a time series forecasting to predict the unused resource for long-term service jobs which is not suitable for short-lived jobs, and the prediction accuracy relies on the existence of patterns in the training data, which can increase the prediction error rate and thus increase the chance of over-provisioning, decreasing the resource utilization. Also, RCCR does not adequately consider the fluctuations of the unused resource in short-lived jobs, which can increase the error rate and thereby increase the probability of over-provisioning. The resource utilizations in CORP and RCCR are higher than that in CloudScale and DRA. This is because CORP and RCCR allocate the resource to jobs in an opportunistic approach in which the allocated unused resource can

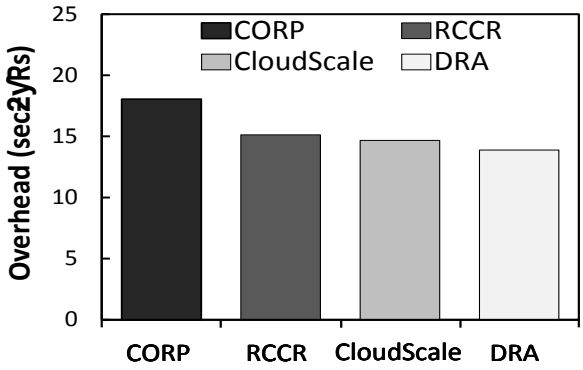


Figure 3.12: Overhead of different methods on a real cluster.

be reallocated to other new arriving jobs with a certain probability, which can increase the resource utilization. DRA has the lowest resource utilization among all the methods because DRA neglects the fluctuations of the resource which can result in inaccurate prediction of the resource and thus may lead to over-provisioning. Also it is a demand-based resource allocation and does not utilize the allocated but unused resource and reallocate it to other jobs to increase the resource utilization.

To test the effects of job packing in increasing resource utilization, we also evaluated the performance of CORPW/oP, a variant of CORP in which job packing is not used. Specifically, we used Equ. (3.1) to calculate the average resource utilization of CPU, MEM and storage with the number of jobs varying from 50 to 300, respectively. Figure 3.8(d) shows the average resource utilization of different resource types in different methods. We see that the average resource utilization follows: CORP>CORPW/oP>RCCR>CloudScale>DRA. The average resource utilization of CORPW/oP is lower than CORP and higher than RCCR. This is because both CORPW/oP and CORP allocate the resource to jobs in an opportunistic approach, which can increase the resource utilization. Also, CORPW/oP and CORP use deep learning to predict the unused resource and adequately consider the fluctuations of unused resource by using the HMM model to correct the prediction error, and thereby decreases the resource over-provisioning. However, CORPW/oP does not leverage complementarity of jobs' demands on different resource types and utilize the job packing strategy to reduce the resource fragmentation, which is considered in CORP. Thus, CORPW/oP has lower average resource utilization than CORP. Also, the result RCCR>CloudScale>DRA is consistent with that in Figures 3.8(a)-3.8(c) due to the same reasons.

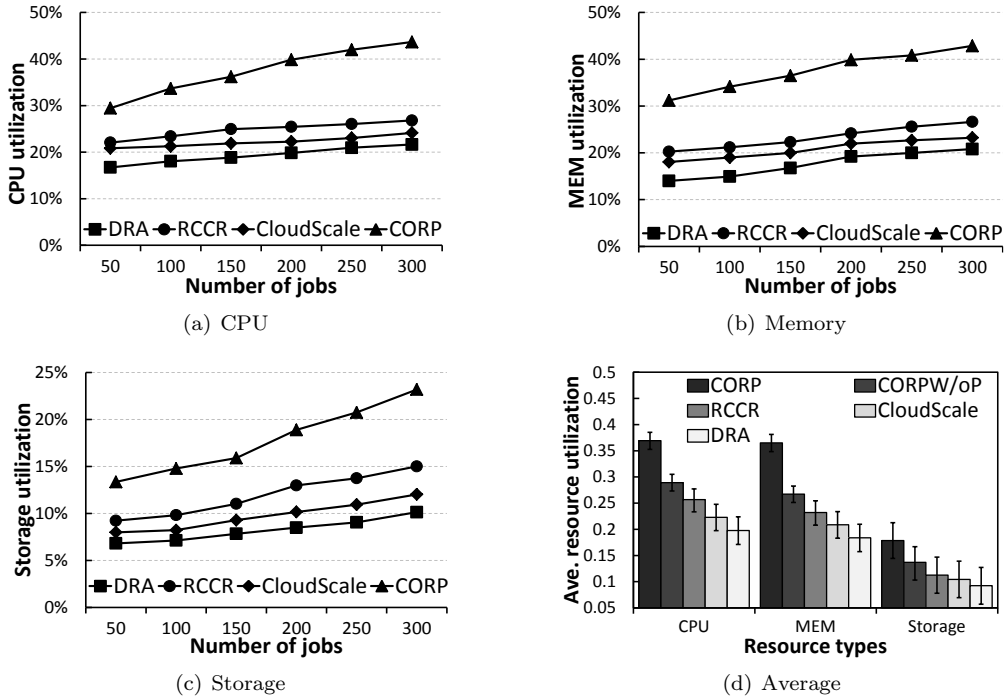
We regard DRA as a baseline and calculate the ratio of the resource wastage reduction of each of the other methods. Specifically, we first used Formula (3.3) to calculate the resource wastage

ratio of the resource type, then we calculate the resource reduction of each method, finally we regard DRA as a baseline. Figure 3.9 shows the relationship between the resource wastage reduction ratio of different methods. We see that the result follows CORP>RCCR>CloudScale>DRA, which is consistent with the result in Figure 3.8. The reasons are similar to that explained in Figure 3.8.

We used Equ. (3.2) to calculate the overall resource utilization (the weighted average of the utilizations of CPU, MEM and storage). Compared to CPU and MEM, storage is not the bottleneck resource, hence we set the weights for CPU, MEM and storage as 0.4, 0.4 and 0.2, respectively. We varied the SLO violation rate by varying the probability threshold P_{th} and thereby varying the percentage of jobs that have SLO violation. Specifically, we considered the SLO violation occurs when a job's response time exceeds the threshold on its response time (We assume jobs' response time is affected by the unavailability of resource for job processing [104].). We recorded the overall resource utilization when the SLO violation rate (approximately) equals 5%, 10%, 15%, 20%, 25% and 30%. Figure 3.10 shows the relationship between the overall resource utilization and the SLO violation rate. We find that the overall resource utilization increases as the SLO violation rate increases. This is because the larger the SLO violation rate, the lower the probability that the resource over-provisioning occurs and thus the higher the overall resource utilization. Also, we see that given an SLO violation rate, the overall resource utilization follows CORP>RCCR>CloudScale>DRA due to the same reasons in Figure 3.8.

Figure 3.11 shows the relationship between the SLO violation rate and the confidence level on a real cluster. From Figure 3.11, we find the SLO violation rate decreases as the confidence level increases. This is because the higher the confidence level, the more conservative the prediction, and the less the amount of resource that will be allocated to jobs in the risk of SLO violations. Also, we find that the SLO violation rate follows CORP<RCCA<CloudScale<DRA due to the same reasons in Figure 3.8. RCCA uses a time-series based forecasting to predict the unused resource with confidence interval prediction and error correction, which can decrease SLO violation probability. CloudScale uses a prediction error handling to correct prediction errors and perform online adaptive padding to avoid overestimation errors. However, DRA does not have a strategy to handle prediction errors.

We evaluated the overhead of different methods by measuring the latency for allocating resource to 300 jobs in each method. Figure 3.12 shows the latency of different methods on a real cluster. In Figure 3.12, we see that the latency of CORP is slightly higher than the other methods.



(a) CPU (b) Memory

(c) Storage (d) Average

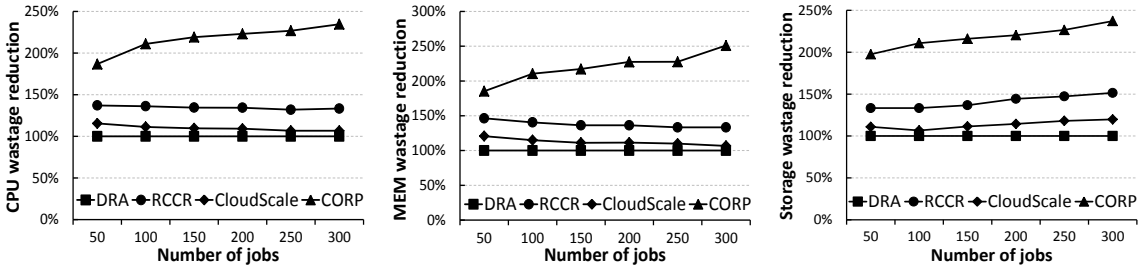
Figure 3.13: Utilizations of different resource types vs. number of jobs of different methods on Amazon EC2.

This is because CORP uses the deep neural network to predict the amount of unused resource of jobs. The deep neural network has complex structure with multiple layers, which obtains accuracy at the expense of computation overhead and thus increases the latency a little [69]. However, the other methods do not have such complex structure for prediction, thus they have relatively lower latency.

3.4.2 Experimental Results on Amazon EC2

To further verify the performance of CORP, we also compared CORP with other methods on Amazon EC2. The servers are from commercial product HP ProLiant ML110 G5 servers (2660 MIPS CPU, 4GB memory) [37]. Each server is set to have 1GB/s bandwidth and 720GB disk storage capacity.

Figure 3.13 shows the relationship between the resource utilization and the number of jobs on Amazon EC2. Similarly, we see the resource utilization increases as the number of jobs increases, and the resource utilization follows CORP>RCCR>CloudScale>DRA due to the same reasons explained in Figure 3.8. By examining Figures 3.13(a)-3.13(c), we see that the utilizations of CPU and MEM



(a) CPU (b) Memory (c) Storage

Figure 3.14: Resource wastage reduction vs. number of jobs of different methods on Amazon EC2.

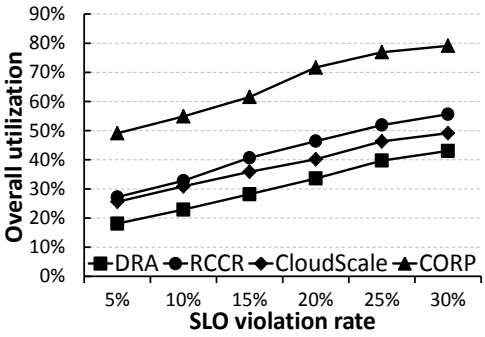


Figure 3.15: Resource utilization vs. SLO violation rate on Amazon EC2.

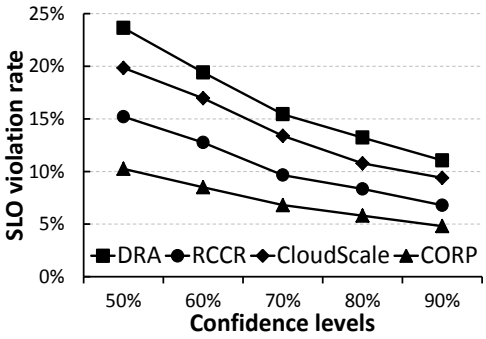


Figure 3.16: SLO violation rate vs. confidence levels on Amazon EC2.

are higher than storage. This is because the storage is not the bottleneck resource and has more wastage in allocation compared to CPU and MEM, thereby has lower resource utilization. We also tested the effects of job packing in increasing resource utilization on Amazon EC2. Figure 3.13(d) shows the average resource utilization of different resource types. In Figure 3.13(d), we also see that the average resource utilization follows: CORP>CORPW/oP>RCCR>CloudScale>DRA due to the same reasons explained in Figure 3.8(d).

We regarded DRA as a baseline and calculated the ratio of the resource wastage reductions of RCCR, CloudScale and CORP. Specifically, we first used Equ. (3.3) to calculate the resource wastage ratio of the resource type, then we calculated the resource reduction of each method, finally we regarded DRA as a baseline. Figure 3.14 shows the relationship between the resource wastage reduction ratio of different methods. We see that the result follows CORP>RCCR>CloudScale>DRA, which is consistent with the results in our experiment on the real cluster in Figure 3.8 due to the same reasons.

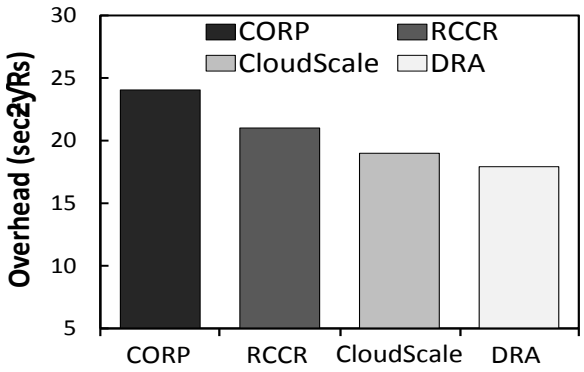


Figure 3.17: Overhead of different methods on Amazon EC2.

Figure 3.15 shows the relationship between the overall resource utilization and the SLO violation rate on Amazon EC2. From Figure 3.15, we also find the overall resource utilization increases as the SLO violation rate increases and given an SLO violation rate, the overall resource utilization follows CORP>RCCR>CloudSCale>DRA due to the same reasons explained in Figure 3.8.

Figure 3.16 shows the relationship between the SLO violation rate and the confidence level on Amazon EC2. We also find that given a confidence level, the SLO violation rate decreases as the confidence level increases and the SLO violation rate follows CORP<RCCA<CloudScale<DRA due to the same reasons explained in Figure 3.11.

Figure 3.17 shows the overhead of different methods measured by the latency for allocating resource to 300 jobs on Amazon EC2. Figure 3.17 mirrors Figure 3.12 due to the same reasons. Comparing Figure 3.17 and Figure 3.12, we see that the latency in Figure 3.17 is relatively higher than that in Figure 3.12. This is because the communication overhead in Amazon EC2 is relatively higher than that in the cluster.

Our experimental results based on the real cluster and Amazon EC2 show that CORP has the best overall performance. This is because CORP explores the potential tradeoff between the efficiency and computation overhead by using deep learning and HMM to get high prediction accuracy of unused resource and utilizing job packing together to obtain high resource utilization.

3.5 Conclusion

This chapter presents a cooperative opportunistic resource provisioning scheme (CORP) for short-lived jobs, which offers the temporarily-unused resource in an opportunistic manner.

Specifically, this chapter first introduces the background of deep learning. Next, it introduces the cooperative opportunistic resource provisioning problem, in which the deep learning algorithm is used to predict the amount of allocated and unused resources of short-lived jobs without resource usage patterns and the HMM model is used to predict the fluctuations of amount of unused resource caused by jobs' time-varying resource demands so that the prediction error can be corrected. Then, the chapter describes the design of CORP, which utilizes the job packing strategy to allocate the unused resource to jobs. Finally, the chapter describes the experimental results based on a real cluster and Amazon EC2, which show that CORP achieves high resource utilization and provides high SLO guarantee.

Chapter 4

Dependency: A Non-negligible Factor in Scheduling and Preemption for High Throughput in Clouds

In this chapter, we introduce our dependency-aware scheduling and preemption strategy (DSP). We first introduce the system model of the DSP, in which the makespan minimization model is illustrated. Next, we describe the strategy of dependency-considered task priority determination. Then we present the probabilistic based preemption approach. Experimental results based on a real cluster and Amazon EC2 cloud service show that DSP achieves high throughput, low overhead and job satisfaction compared to existing scheduling and preemption strategies.

4.1 System Model

In a distributed parallel computing system, there are workers and schedulers. Workers are used to execute tasks, and schedulers are used to assign tasks to workers. We assume a job is split into m tasks (jobs do not overlap in tasks), and the tasks are allocated to workers. Each worker has

a buffer queue which is used for queuing tasks when a worker is allocated to more tasks than it can run concurrently.[1]

Service time: The time that a task spends executing on a worker machine.

Job response time: The time from when a job is submitted to the scheduler until the tail task of the job finishes execution.

Makespan: The makespan (or schedule length) is the time when all jobs finish execution.

Throughput: The total number of jobs that complete their executions within the job deadlines per unit of time.

4.1.1 Dependency-considered Scheduling

We consider a scheduling problem with the objective of scheduling multiple jobs onto multiple heterogeneous workers to minimize the schedule length (makespan), and thus maximize the throughput. Task dependency determines the order of tasks' execution, and a task depending on another task cannot start executing before the other task completes. Scheduling tasks to workers without the consideration of dependency incurs dependency violation, which delays tasks' completion time and decrease the throughput. By simply considering dependency in scheduling, it cannot increase the chance that more tasks can start execution after the task that has more dependent tasks finishes execution. This can also increase the makespan and thus decrease the throughput. Below, we present an example to explain it. Figure 1.2 illustrates the diverse dependency relations among tasks. Tasks T_2-T_5 cannot start executing until task T_1 finishes its execution. Similarly, tasks T_7-T_{14} and T_{16}-T_{19} cannot start executing until T_6 and T_{15} finish their executions, respectively. Executing the task at first which has more dependent tasks helps increase the throughput. Hence, dependency consideration is important and necessary in task scheduling, which directly affects the throughput [118]. Unlike previous works, DSP substantially takes into account task dependency and fully utilizes task dependency information and judiciously determines tasks' execution order to increase throughput. Also, DSP schedules tasks with the consideration of job deadlines neglected in the work [57], which can increase the throughput. Unlike previous works, DSP takes into account task dependency and assigns tasks that are independent of each other to different workers (or processors) so that the tasks can run in parallel, and thus decreases the makespan. DSP fully utilizes task

[1] Although some cluster schedulers do not support buffer queues, we assume a worker has a buffer queue [32,47,90] for the purpose of utilizing task dependency information to increase throughput in scheduling and preemption of cloud-scale computing clusters [32].

dependency information and judiciously determines tasks' execution order so that more tasks have chance to start executing after a selected task finishes execution, also, DSP schedules tasks with the consideration of job deadlines neglected in the work [57], which can increase the throughput.

Denote $\mathcal{L}_{\mathcal{MS}}$ as the makespan for all jobs submitted at time t. Suppose h jobs ($\mathcal{J} = \{J_1, ..., J_h\}$) are submitted at time t. The deadlines on completion time of h jobs are represented by $t_1^d, ..., t_h^d$, respectively. Denote t_{ij}^s as the starting time of task T_{ij}, and t_{ij}^e as the ending time (completion time) of task T_{ij}. Let $C_\imath = \{T_{ij}(1), ..., T_{ij}(|C_\imath| + 1)\}$ be an arbitrary chain of tasks belonging to job J_i ($i \in \{1, ..., h\}$), representing the dependency of the tasks, where $|C_\imath|$ represents the length of the chain $C_\imath$. All tasks $T_{ij} \in C_\imath$ ($i \in \{1, ..., h\}, j \in \{1, ..., m\}$) must be processed

Table 4.1: Notations in DSP.

$\mathcal{J}$	A set of jobs	$C_\imath$	A chain of tasks
h	# of jobs in $\mathcal{J}$	$\|C_\imath\|$	The length of $C_\imath$
J_i	The ith job in $\mathcal{J}$	$g(k)$	Proc. rate func. of worker k
t_i^d	Job J_i's deadline	l_{ij}	Task T_{ij}'s size
T_{ij}	The jth task of J_i	$\mathcal{L}_{\mathcal{MS}}$	Makespan
t_{ij}^s	T_{ij}'s starting time	$\mathcal{M}$	A set of target workers
t_{ij}^e	T_{ij}'s ending time	$t_{ij,k}$	T_{ij}'s Exec. time on worker k
t_{ij}^r	T_{ij}'s rec. time / preemption	n	Total # of workers
s_{mem}^k	Mem. size of worker k	m	# of tasks / job
s_{cpu}^k	CPU size of worker k	N_{ij}^p	# of preemptions for T_{ij}
t_{ij}^{rem}	T_{ij}'s remaining time	P_{ij}^t	T_{ij}'s priority at time t

sequentially one after another. Task $T_{ij}(k+1)$ cannot start executing until task $T_{ij}(k)$ finishes execution, where $k \in \{1, ..., |C_\imath|\}$. Denote $t_{ij}^s(k)$ as the starting time of the kth task on the chain $C_\imath$ belonging to job J_i running on a worker machine, and $t_{ij}^e(k)$ as the ending time of the kth task running on a worker machine. Thus, for a chain $C_\imath$ of tasks belonging to job J_i, the $(k+1)$th task $T_{ij}(k+1)$ on $C_\imath$ cannot start executing until the kth task $T_{ij}(k)$ on $C_\imath$ finishes its execution, $\sum_{T_{ij} \in C_\imath} \sum_{k=1}^{|C_\imath|} t_{ij}^s(k+1) - t_{ij}^e(k) \geq 0$. For easy reference, Table 4.1 shows the main notations of DSP in this dissertation.

Denote $x_{ij,k} = \{0, 1\}$ as an indicator variable representing if task T_{ij} is assigned to worker k, $x_{ij,k} = 1$ if T_{ij} is assigned to worker k, otherwise $x_{ij,k} = 0$. Define $g(k)$ ($k \in \{1, ..., n\}$) as a function of processing rate of the kth worker machine, which is the million instructions per second (MIPS) speed, and $g(k)$ is related to the CPU size s_{cpu}^k and memory size s_{mem}^k of the worker k [45]. The larger the CPU size, the higher the processing rate; the larger the memory size, the higher the

processing rate. Specifically, the function $g(k)$ is expressed as follows

$$g(k) = \theta_1 s_{cpu}^k + \theta_2 s_{mem}^k \tag{4.1}$$

where θ_1 and θ_2 are the weights for CPU size and memory size, respectively. Denote $t_{ij,k}$ as the time for executing task T_{ij} on worker k without being preempted. Thus, we have

$$t_{ij,k} = \frac{l_{ij}}{g(k)} \tag{4.2}$$

where l_{ij} is the size of task T_{ij} in terms of Millions of Instructions (MI) [10]. However, suspending a running task and putting a waiting task to running state consumes a certain amount of time (called recovery time) due to the context switching. Let t_{ij}^r be the recovery time of a preemption for task T_{ij}, and N_{ij}^p be the number of preemptions for task T_{ij}. Denote σ as the threshold for the time (i.e., 0.05s) that an *evicted* task waits for execution, and we approximately use it as the waiting time of an evicted task. Let $y_{ij,uv,k} = 1$ if task T_{ij} precedes task T_{uv}, otherwise $y_{ij,uv,k} = 0$. Since linear programming (LP) is considered more efficient in solving scheduling problems [60], we formalize our problem as the following linear optimization problem [70]:

$$Min\{\mathcal{L}_{\mathcal{MS}}\} \tag{4.3}$$

$$\begin{aligned} s.t.\ t_{ij}^s + \sum_{k=1}^{n} t_{ij,k} \cdot x_{ij,k} + \sum_{k=1}^{n} N_{ij,k}^p \cdot (t_{ij}^r + \sigma) \cdot x_{ij,k} \leq \mathcal{L}_{\mathcal{MS}} \\ (\forall i \in \{1, ..., h\}, j \in \{1, ..., m\}, k \in \{1, ..., n\}) \end{aligned} \tag{4.4}$$

$$(t_{ij}^s + t_{ij,k}) \cdot x_{ij,k} \leq (t_{uv}^s + (1 - y_{ij,uv,k}) \cdot M) \cdot x_{uv,k}\ (\forall u \in \{1, ..., h\}, v \in \{1, ..., m\}) \tag{4.5}$$

$$t_{ij}^s + \sum_{k=1}^{n} t_{ij,k} \cdot x_{ij,k} + \sum_{k=1}^{n} N_{ij,k}^p \cdot (t_{ij}^r + \sigma) \cdot x_{ij,k} \leq t_i^d\ (T_{ij} \in C_\imath) \tag{4.6}$$

$$t_{ij}^s + \sum_{l=1}^{n} t_{ij,l} \cdot x_{ij,l} + \sum_{l=1}^{n} N_{ij,l}^p \cdot (t_{ij}^r + \sigma) \cdot x_{ij,l} \leq t_{iq}^s \ (q \in \{1, ..., m\}, l \in \{1, ..., n\}) \tag{4.7}$$

$$y_{ij,uv,k} + y_{uv,ij,k} = 1 \tag{4.8}$$

$$y_{ij,uv,k} \in \{0, 1\} \tag{4.9}$$

$$x_{ij,k} \in \{0, 1\} \tag{4.10}$$

$$t_{ij}^s \geq 0 \tag{4.11}$$

where M in Formula (4.5) is a very large positive real number.

The constraint (4.4) is to ensure the time consumed by the job completing at last is no longer than the makespan. The constraint (4.5) is to ensure the execution order between T_{ij} and T_{uv} on worker k. The constraint (4.6) is to ensure jobs can finish execution within their specified deadlines.[2] The constraint (4.7) is to ensure the dependency relation between T_{ij} and T_{iq}, that is, task T_{ij} precedes task T_{iq} on worker k. The output of the LP model includes the set $\mathcal{M} = \{k | x_{ij,k} = 1, \forall i \in \{1, ..., h\}, j \in \{1, ..., m\}, k \in \{1, ..., n\}\}$ and the starting time of tasks' executions t_{ij}^s $(\forall i \in \{1, ..., h\}, j \in \{1, ..., m\})$. The tasks are assigned to their target workers indicated in the set $\mathcal{M}$ according to their starting times. The target worker for T_{ij} is the kth worker when $x_{ij,k} = 1$. The set $\mathcal{M}$ is a set (multiset) of workers, which represents the target workers for all the tasks T_{ij} $(\forall i \in \{1, ..., h\}, j \in \{1, ..., m\})$ derived from the LP model. In Inequs. (4.4), (4.6), (4.7), the number of preemptions is related to the task's size and priority. We adopt a checkpoint-restart mechanism from [49] (tasks are restarted from their most recent checkpoints) to compute the optimal number of checkpointing intervals, and estimate the number of preemptions for each task using historical data. We use the number of checkpoints as an upper bound for the number of preemptions of a task. Based on the task duration and average checkpointing cost, we calculate the number of preemptions for a given task length.

[2] With the historical data, the execution time of a task can be estimated with a good accuracy [47,62,63]. Although it is hard to give a high accurate estimation of a task's execution time, our approach works well with an upper bound of a task's execution time.

The target worker $k|_{x_{ij,k}=1}$ ($i \in \{1, ..., h\}, j \in \{1, ..., m\}, k \in \{1, ..., n\}$) for task T_{ij} must be an integer in practice. To make our formulated optimization problem more tractable, we can first relax the problem to a real-number optimization problem in which $k|_{x_{ij,k}=1}$ can be a real number, and derive the solution for the real-number optimization problem. Then, we can use integer rounding to get the solution for practical use. Considering the task scheduling problem in general is NP-complete [118], we use the CPLEX linear program solver [5] to solve this linear optimization problem.

Given a set of jobs consisting of tasks, constraints of jobs and tasks (i.e., task precedence constraints, job completion time constraints), and a number of heterogeneous worker machines, the output of the problem solution provides the task schedule $[t^s_{ij},\ k|_{x_{ij,k}=1}]$ ($\forall i \in \{1, ..., h\}, j \in \{1, ..., m\}, k \in \{1, ..., n\}$), that is, the target worker (denoted by $k|_{x_{ij,k}=1}$) and the starting time (denoted by t^s_{ij}) for each task. Then, the minimum makespan (denoted by $\mathcal{L}_{\mathcal{MS}}$) can be obtained from the LP model. Based on the output of the LP model $[t^s_{ij},\ k|_{x_{ij,k}=1}]$ the schedulers assign tasks to their target workers by following the order of their starting times.

4.2 Dependency-considered Task Priority Determination

It has been shown that assigning higher priority to shorter remaining task increases the throughput. However, this also increases the waiting time of large size tasks. So, it is desired to consider waiting time to avoid starvation of large size tasks to reduce the average waiting time for each task. Thus, previous preemption methods define task priority based on task remaining time and (or) waiting time and always run the task with the highest priority. However, they neglect dependency in the priority determination in preemption, which is, however, an important factor to consider in preemption to maximize the throughput. Even if they select the task with the highest priority among the runnable tasks, the throughput cannot be fully increased. Because choosing a task with more dependent tasks to run enables more tasks to be runnable next, and a better task that can more increase the throughput can be selected from more options. In this section, we propose task priority that considers task dependency to achieve a higher throughput.

Figure 4.1 shows an example illustrating the importance of considering dependency in preemption. In Figure 4.1, there are 7 tasks $T_1, ..., T_7$. T_2 and T_3 depend on T_1; T_4 and T_5 depend on T_2; T_6 and T_7 depend on T_3. DSP assigns higher priority to tasks on which more tasks depend. This

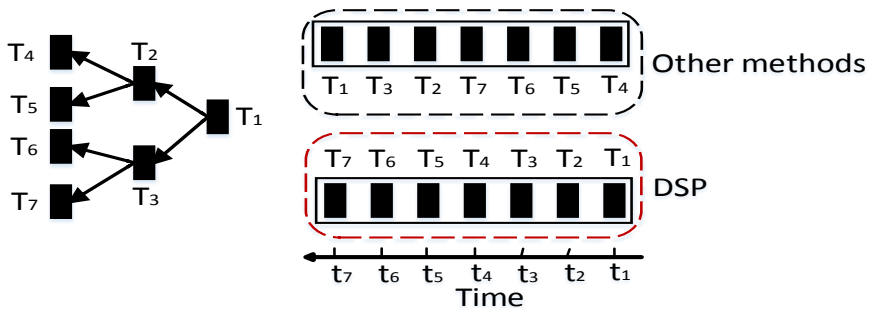


Figure 4.1: Determining task priority by considering task dependency.

helps increase the throughput by increasing the chance that more tasks can start executing after the precedent task finishes execution. Without considering task dependency, other methods may assign priorities to tasks by following: $T_1<T_3<T_2<T_7<T_6<T_5<T_4$. In this case, T_1 has the lowest chance to be executed. But all the other tasks cannot start execution if T_1 does not finish execution, and even worse, deadlock may occur due to the dependency constraints, which can increase tasks' waiting time and decrease the throughput. With dependency consideration, task remaining time and waiting time in priority determination, DSP assigns priorities to tasks by following: $T_7<T_6<T_5<T_4<T_3<T_2<T_1$ or $T_6<T_7<T_5<T_4<T_3<T_2<T_1$, etc.[3] Thus, DSP helps increase throughput.

Below, we first show an example to indicate the importance of considering different kinds of dependencies in preemption, and then propose a method to determine task priority in order to increase the throughput while reducing the average waiting time for each task. If a task has more dependent tasks, then after the task finishes execution, more tasks become runnable tasks and a better task can be chosen to more increase the throughput. Similarly, assigning higher priority to the task that has more dependent tasks can increase the chance that more tasks depending on it can start executing after the task finishes execution. As shown in Figure 4.2, there are different kinds of dependencies [93,122]. Although T_1 and T_6 have the same number of dependent tasks, T_6 has higher priority than T_1 for execution because more tasks can be executed soon after T_6 finishes execution and a better option can be chosen from the candidates. Similarly, T_{11} should have a higher priority than T_6. Task T_{11} has more dependent tasks compared with the other tasks (e.g., T_1, T_6). Although T_{11} and T_6 have the same number of dependent tasks in the first level which is more than T_1, T_{11} has more dependent tasks in the second level than T_6. In this case, executing T_{11} at first helps increase the throughput. Thus T_{11} has higher priority than other tasks from the perspective of considering

[3]To increase the throughput, DSP utilizes task dependency information to preschedule tasks (including tasks that are ready to run) [12].

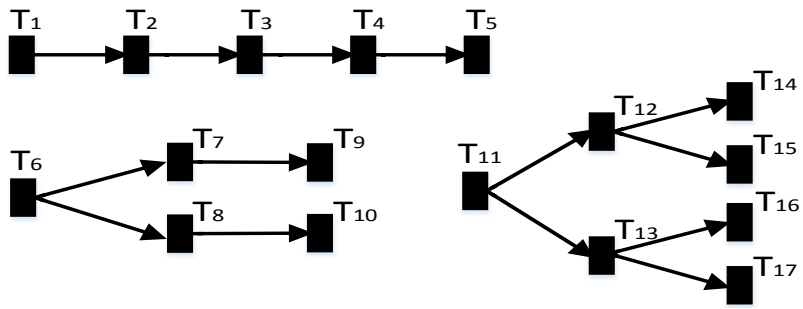


Figure 4.2: Utilize task dependency to determine task priority.

dependency. Similarly, T_6 has higher priority than other tasks (except T_{11}), such as T_{12} and T_{13} from the perspective of task dependency consideration.

To increase throughput, we also consider assigning higher priority to a task with less remaining time. Based on the queuing theory, choosing a task with shortest remaining time to preempt the running task can increase the throughput [38, 102]. Thus we take into account tasks' remaining time to determine task priority. Since shortest remaining first can easily incur the starvation of tasks with longer remaining time, we also consider tasks' waiting time to determine task priority for the avoidance of starvation to increase the throughput.

The rationale behind our design of task priority determination can be summarized as follows: The reason for utilizing task dependency to recursively calculate a task's priority based on its children's priorities (see Formula (4.12)) results from deadlock avoidance in operating system design.

Task's Remaining Time: From the queuing theory, it is known that Shortest-Remaining-Processing-Time-first (SRPT) minimizes the queuing time [102]. Allowing short remaining time task to preempt long remaining time task is desirable because forcing long remaining time tasks to wait behind short remaining tasks generates much lower mean response time than the case in which short remaining tasks must wait behind long remaining tasks. Based on this, we assign higher priority to those tasks with short remaining time.

Waiting Time: Although SRPT method has the advantage of minimizing the queuing time in the worker, it can starve long remaining time tasks. Based on this, we also consider the waiting time of a task to determine the task's priority. The priority of a task increases as its waiting time increases.

Denote t_{ij}^{rem} as the remaining time of task T_{ij}, t_{ij}^{w} as the waiting time of task T_{ij}. To determine the priority of a task T_{ij}, we consider it as a function of its remaining time t_{ij}^{rem} and its waiting time t_{ij}^{w} based on the dependency relations among T_{ij} and its children (i.e., the tasks

depending on T_{ij}).

Given a particular and arbitrary task T_{ij}, the priority of task T_{ij} at time t is recursively computed as follows:

$$P_{ij}^{t} = \sum_{T_{ik} \in C_i} (\gamma + 1) P_{ik} \tag{4.12}$$

where C_i represents a set consisting of T_{ij}'s children, and $\gamma \in (0, 1)$ is a certain coefficient. For a given task (e.g., T_{ij}) that has no dependent tasks, the priority of task T_{ij} at time t is computed in below:

$$P_{ij}^{t} = \omega_1 \cdot \frac{1}{t_{ij}^{rem}} + \omega_2 \cdot t_{ij}^{w} \tag{4.13}$$

where ω_1 and ω_2 are the weights for the task's remaining time and waiting time, respectively, and $\omega_1 + \omega_2 = 1$. The priority of task T_{ij} at time t_1 ($P_{ij}^{t_1}$) can be obtained based on Formulas (4.13) and (4.12).

4.3 Probabilistic Based Preemption

Traditional preemption methods preempt a running task by simply comparing the priorities of the waiting task and the running task, and the preemption occurs if the waiting task's priority is higher than the running task (see Figure 4.3).

Consider a case that high priority tasks arrive frequently, then the preemption occurs frequently. This incurs much time overhead for context switching and increases the average waiting time of jobs (especially low priority jobs) and thus may increase job completion time and decrease the throughput. We use an example to illustrate this case. Suppose the priority of the current running task T_1 is 1.1, when a task (T_2) with priority 1.7 arrives, then T_2 preempts T_1. Soon after T_2 starts running, another task T_3 with priority 2 arrives at the waiting queue, then T_3 preempts T_2. Such frequent preemptions lead to many context switches, which extends job completion time and possibly reduces throughput. Since T_2 and T_3 have close priority, the throughput loss caused by preemption may offset or may be even higher than the throughput increase caused by running T_3 first. In such case, the preemption is not necessary. On the other hand, if the running task has much higher priority than all waiting tasks, it may greatly increase the waiting time of the waiting

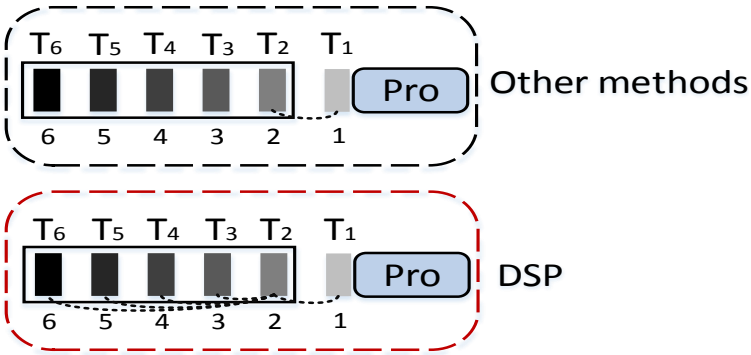


Figure 4.3: Preemption decision in different methods.

tasks.

In order to avoid too frequent preemptions and too infrequent preemptions to achieve high throughput and low waiting time for each task, we propose a probabilistic based preemption method. It chooses a task among the waiting tasks with the top priority range with the probability equaling the task's normalized priority. The normalized priority of a task is defined as the ratio of the reverse rank of the task's priority among all the tasks in the queue to the number of tasks in the queue (i.e., $\frac{k-\sum_{j=i+f}^{i+f+k-1} I_{i+f,j}}{k}$). With this method, in the above example, if we consider the priority rank of the waiting task among all the waiting tasks to choose a task with a higher priority rank, T_2 might not preempt T_1 even though T_2's priority is a little higher than T_1. Also, if the running task has run for a too long time, it will be preempted by a high-priority task. As a result, a preemption is unlike to occur if its time overhead offsets the throughput gains by the preemption. Also, the waiting time of tasks is reduced.

4.3.0.1 Probabilistic based Preemption

Given a worker machine m_i, without loss of generality, suppose there are f running tasks $(T_i, ..., T_{i+f-1})$, and there are k tasks waiting $(T_{i+f}, ..., T_{i+f+k-1})$ in the queue of m_i (see Figure 4.4).

Given a task, the task can execute only if the following conditions are satisfied:

- Condition C_1: The priority of the waiting task is higher than that of the running task.
- Condition C_2: The waiting task does not depend on the running task.
- Condition C_3: The rank of the waiting task's priority among all of the waiting tasks in the queue is top $\lfloor(1-\delta)k\rfloor$ (based on Theorem 4.1) for each processor.

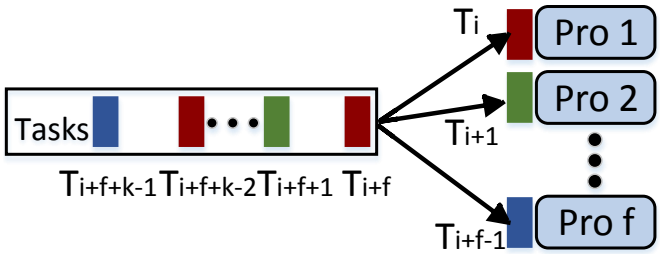


Figure 4.4: Preemption for multiple tasks running on multiple processors.

We calculate the priorities of the running tasks $T_i, ..., T_{i+f-1}$ and the waiting tasks $T_{i+f}, ...,$ $T_{i+f+k-1}$ using Formulas (4.12) and (4.13). Define I as an indicator function. For any two tasks T_i and T_j,

$$I_{ij} = \begin{cases} 1, & P_i > P_j \\ 0, & P_i \leq P_j \end{cases} \tag{4.14}$$

where P_i and P_j are the priorities of task T_i and task T_j.

For a particular and arbitrary running task T_i in worker m_i, we check if Condition C_3 is satisfied using the following formula:

$$\frac{\sum_{j=i+f}^{i+f+k-1} I_{i+f,j}}{k} \geq \delta \tag{4.15}$$

where δ is the minimum required ratio between 0 and 1.

To understand how Formula (4.15) works, we give an example to illustrate the process of selecting a waiting task to preempt the running task. Suppose there are 10 tasks $(T_1, ..., T_{10})$ in the waiting queue of a worker, and their priorities are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. According to Formula (4.15), the priority of the waiting task should be greater than or equal to $\lceil 10\delta \rceil$ other waiting tasks. Also the waiting task should be independent of the running tasks, thus at most $\lfloor 10(1-\delta) \rfloor$ tasks are allowed for preemption. If $\delta = 0.5$, then at most top 5 priority tasks are allowed for preemption.

Theorem 4.1. *Given a server with f processors, if f tasks $(T_i, ..., T_{i+f-1})$ are running, and k tasks $(T_{i+f}, ..., T_{i+f+k-1})$ are waiting in the queue of the server, then at most the top $\lfloor (1-\delta)k \rfloor$ priority tasks are allowed for preemption for each processor.*

Proof: According to Formula (4.15), at least $\lceil \delta k \rceil$ tasks' priorities are lower than P_{i+f} if the preemption is allowed for the task T_{i+f}, that is, it allows preemption for at most the top $k - \lceil \delta k \rceil$

Algorithm 8: Pseudocode for Probabilistic based Preemption

```
Input: k tasks T_{i+f}, ..., T_{i+f+k-1} are in the queue of worker m_i waiting for the
       running task(s) T_i, ..., T_{i+f-1}. T^W = [T_{i+f} ··· T_{i+f+k-1}], T^R = [T_i ··· T_{i+f-1}],
       A = [P_{i+f} ··· P_{i+f+k-1}], B = [P_i ··· P_{i+f-1}]
1  Compute the priorities P_i, P_{i+1}, ..., P_{i+f+k-1}
2  Sort (A)   //Descending order
3  Sort (B)   //Ascending order
4  for each i = 1, ..., k do
5      sum[i] ← 0
6  for each i = 1, ..., k do
7      for each j = 1, ..., k do
8          Compute I_{i,j}   //Formula (4.14)
9          sum[i] ← sum[i] + I_{i,j}
10 Pre ← False; low ← 0; high ← 0
11 for each i = 1, ..., k do
12     for each j = 1, ..., f do
13         if T^W[i] depends on T^R[j] then
14             j ← j + 1
15         else
16             if t^w[i] > τ then //The waiting time of task T^W[i] is longer than
               threshold τ
17                 Pre ← True; low ← j; high ← i
18                 break
19             else
20                 if sum[i]/k ≥ δ then //Formula (4.15)
21                     if B[j] < A[i] then
22                         Pre ← True; low ← j; high ← i
23                         break
24                 else
25                     i ← i + 1
26 if Pre = True then
27     Suspend task T^R[low]
28     Run task T^R[high]
29 return Pre
```

tasks for each processor. Thus at most the top $\lfloor(1-\delta)k\rfloor$ priority tasks are allowed for preemption.

Therefore, Theorem 4.1 holds. ■

Algorithm 8 shows the pseudocode for periodically[4] selecting waiting tasks to preempt the running tasks using PP. The algorithm first computes the priorities of all running tasks and waiting tasks (line 1). It then chooses a waiting task from the top $\lfloor(1-\delta)k\rfloor$ priority waiting tasks with the probability equaling its normalized priority (i.e., $\frac{k-\sum_{j=i+f}^{i+f+k-1} I_{i+f,j}}{k}$) (line 10 to line 25). In the process of selecting waiting tasks to preempt running tasks, the algorithm compares the selected task with the lowest priority running task and determines if the task should be preempted based on the conditions C_1, C_2 and C_3 (line 21, line 13, line 20). If all these conditions are satisfied, then the algorithm suspends the running task and runs the selected waiting task. Otherwise, the algorithm chooses the second lowest priority running task and determines if the running task should be preempted based on the conditions C_1, C_2 and C_3, and so on. However, if the waiting time of a task is longer than the threshold τ (e.g., 0.05s), the algorithm also chooses the task to preempt the running task no matter the conditions C_1 and C_3 are satisfied or not (line 16 to line 18).

4.4 Performance Evaluation

In this section, we introduce our experimental results on a large-scale real cluster Palmetto [9], which is Clemson University's primary high-performance computing (HPC) resource , and Amazon EC2 [2], respectively. To show the performance of our method on throughput or makespan, we compared DSP with Tetris [57]. Tetris shares the same objective of maximizing the throughput (or minimize makespan) and speed up job completion as the DSP does. To test our method's performance on preemption, we compared the results of our method and three other methods Amoeba [19], Natjam [40] and SRPT [22] in various scenarios. Amoeba is preemption based method, which considers task's requirement on resource as task's remaining time and chooses the task that consumes the most resources to be preempted. Natjam tries to utilize different task eviction policies to achieve low completion time for higher priority jobs. Natjam first classifies jobs into production jobs and research jobs, and it assigns higher priority to production jobs and lower priority to research jobs. SRPT attempts to maximize total throughput and minimize average waiting time while balancing load across available grid resources. All these three methods aim at increasing the throughput,

[4]To avoid frequent preemptions, DSP waits for a certain period of time so that there are a certain number of tasks in the waiting queue, and executes Algorithm 8 periodically.

Table 4.2: Parameter settings in DSP.

Parameter	Meaning	Setting
n	# of servers	30-50
h	# of jobs	150-750
m	# of tasks / job	100-2000
δ	Minimum required ratio	0.35
τ	A threshold for waiting time	0.05
θ_1	Weight for CPU size	0.5
θ_2	Weight for Mem size	0.5
α	Weight for waiting time for SRPT	0.5
β	Weight for remaining time for SRPT	1
γ	A certain coefficient $\in (0, 1)$	0.5
ω_1	Weight for task's remaining time	0.6
ω_2	Weight for task's waiting time	0.4

which is consistent with the objective of DSP. Thus we choose these three methods for comparison to show the performance of DSP.

Tetris [57]. Tetris aims to maximize the task throughput (or minimize makespan) and speed up job completion by packing tasks to machines based on their requirements on all the resources. Specifically, when resources on a machine become available, Tetris first selects the set of tasks whose peak usage of each resource can be accommodated on that machine. Then Tetris computes an alignment score (a weighted dot product between the vector of machine's available resources and the task's peak usage of resources) to the machine on which the set of tasks whose peak usage of each resource can be accommodated. The task with the highest alignment score is scheduled and allocated its peak resource demands. This process is repeated recursively until the machine cannot accommodate any further tasks.

In order to show the shortcomings of Tetris and the advantages of our DSP, we also tested the performance of two variants of Tetris: TetrisW/oDep and TetrisW/SimDep. TetrisW/oDep represents the approach that does not consider the dependency relations among different tasks in assigning tasks to workers. TetrisW/SimDep means the scheduler simply considers task dependency and schedules tasks to workers, but it does not utilize task dependency to increase the throughput, that is, the scheduler neglects giving priority to the tasks that have more dependent tasks in scheduling and preemption which is considered in our method.

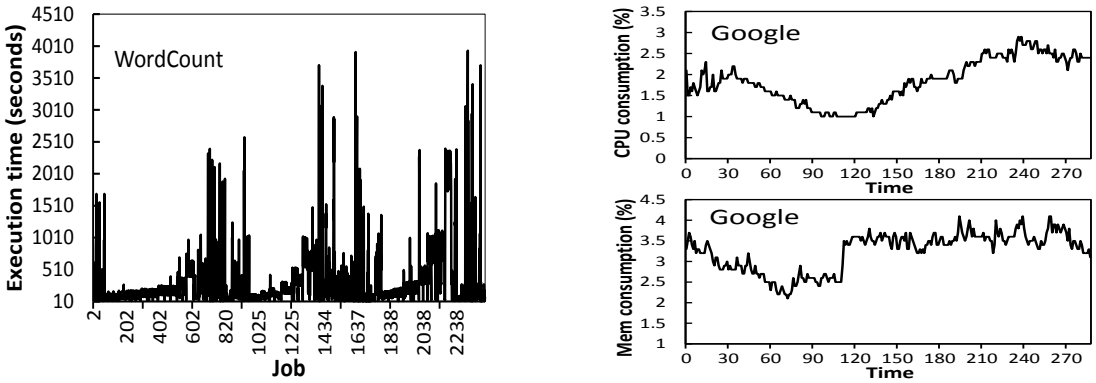
Amoeba [19]. Amoeba aims to enable lightweight elasticity by identifying execution points at which running tasks of over-provisioned jobs can be safely exited, committing their outputs, and spawning new tasks for the remaining work. Amoeba implements opportunistic allocation of spare

resources to jobs, and it chooses the tasks that consume the most resources to be preempted. Amoeba uses a checkpoint/restart mechanism to dynamically change the number of slots assigned to a task, that is, it assigns available slots to tasks, and reclaims slots from tasks by checkpointing their work when there is contention. In the experiment, for Amoeba, we chose the task that consumes the most resource to be preempted. Specifically, we computed the resource consumption by calculating the weighted sum of each resource.

Natjam [40]. Natjam aims to 1) achieve low completion times for higher priority jobs; 2) optimize completion times of lower priority jobs by using an on-demand checkpointing technique that saves the state of a task when it is preempted, so that it can resume where it left off when resources become available and leveraging smart eviction policies that select which low priority jobs and their constituent tasks are affected by arriving high priority jobs. Natjam classifies jobs into production jobs and research jobs. It assigns higher priority to production jobs and lower priority to research jobs. Natjam uses two task eviction policies (Shortest Remaining Time (SRT), Longest Remaining Time (LRT)). It tries to minimize the job completion time for both production and research jobs. In implementation, we combined SRT with LRT to choose task for eviction, and we set the equal weight for both policies for task eviction.

SRPT [22]. SRPT aims to maximize total throughput and minimize average waiting time while balancing load across available resources by backfilling jobs locally and dynamically migrating waiting jobs across nodes to leverage residual resources, while guaranteeing (on a best effort basis) bounded turn-around and waiting times for all jobs. SRPT gives preference to requests for small size files or requests with short remaining file size. Specifically, it uses the linear combination of waiting time and the remaining time for a job (i.e., estimated time for completing the remaining part of the job) to determine the priority of a job. The jobs are queued based on their priorities. In the experiment, we set the weight of waiting time α to 0.5 and the weight of remaining time β to 1 to calculate job priority [22].

We first deployed our testbed in a large-scale real cluster located in our university [9]. We implemented our method and other four methods in our testbed, and evaluated them with 50 servers. We then conducted experiments on the real-world Amazon EC2. We evaluated DSP and the other four methods with 30 servers. The servers are from commercial product HP ProLiant ML110 G5 servers (2660 MIPS CPU, 4GB memory) [37]. In both experiments, we used the Hadoop WordCount trace [61] which comprises of CPU bound MapReduce jobs that calculate the number of occurrences



(a) Execution time of jobs in Hadoop WordCount

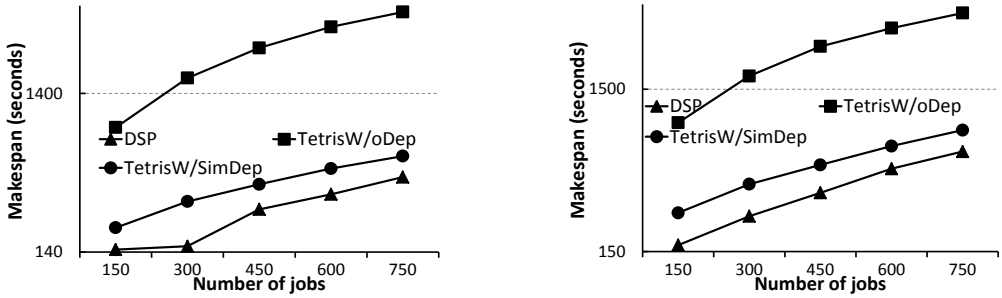
(b) CPU and Mem consumption of a task at different times

Figure 4.5: Analysis of trace data from Hadoop WordCount and Google.

of each word in a text file. We ran MapReduce jobs with input data size 8.2GB. The number of tasks for each job is given by Hadoop WordCount trace. We first chose 150 jobs that can be divided into three categories based on the number of tasks of a job: 2000 tasks for large, 1000 tasks for medium, several hundreds of tasks for small [57] (with equal number of small, medium, and large jobs) to run on the servers. The dependency relations among tasks are generated based on Google Trace [7]. We derived the dependency relations based on the starting time and ending time of tasks. Each server is set to have 1GB/s bandwidth and 720GB disk storage capacity. We set the CPU and memory consumption for each task based on the Google trace [7], and disk and bandwidth consumption for each task are set as 0.02MB [66] and 0.02 MB/s [110], respectively. Figure 4.5(a) shows the execution times of different jobs in Hadoop WordCount. In Figure 4.5(a), we see the execution time of most jobs is less than 1010 seconds. Figure 4.5(b) shows CPU and Mem consumption of a task in Google Trace at different times. In Figure 4.5(b), we find that CPU consumption and Mem consumption of a task vary from 1% to 2.9% and 2.1% to 4.1%, respectively. To fully verify the performances of our method and the other four methods, we varied the number of jobs from 150 to 750 with step size of 150, and repeated the experiment. Table 4.2 shows the parameter settings in our experiment unless otherwise specified.

4.4.1 Effectiveness of Dependency Consideration in Scheduling

Figure 4.6(a) shows the relationship between the makespan and the number of jobs on a real cluster. We see that the makespan increases as the number of jobs increases. This is because with



(a) Performance of various methods on makespan on a real cluster

(b) Performance of various methods on makespan on Amazon EC2

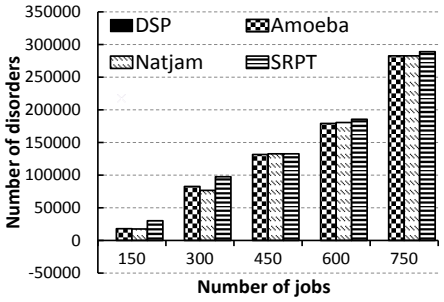
Figure 4.6: Performance of various methods on makespan vs. the number of jobs.

a fixed number of workers, the more jobs submitted, the more time the workers require to finish executing all the jobs. Moreover, we also observe that DSP has the smallest makespan, and the makespans of TetrisW/oDep (Tetris without considering task dependency), TetrisW/SimDep (Tetris with simply considering task dependency) and DSP follow DSP<TetrisW/SimDep<TetrisW/oDep. The reason behind this is that DSP takes into account task dependency and assigns tasks that are independent of each other to different workers (or processors) so that the tasks can run in parallel, and thus decreases the makespan. Also, DSP considers job deadlines and assigns tasks to different workers so that the jobs can finish execution within their deadlines. In addition, DSP uses an LP model to minimize the makespan by fully utilizing task dependency information. Although TetrisW/SimDep considers task dependency, it does not consider job deadline constraints, which can extend job completion time and thus increase the makespan. Also, by simply considering dependency, TetrisW/SimDep cannot fully utilize task dependency information in terms of determining task priority for execution. TetrisW/oDep assigns tasks to workers based on the dot product value of tasks' resource requirements and the available resources of workers, and it may assign tasks that have dependency constraints to different workers. This increases the time overhead for data transmission because the dependent tasks may need to fetch the data (i.e., outputs) of the precedent tasks from different machines. Thus TetrisW/oDep has the longest makespan. We also tested the performance of DSP, TetrisW/oDep and TetrisW/SimDep on Amazon EC2 with 30 workers (shown in Figure 4.6(b)), and we observed similar results. Comparing Figure 4.6(b) and Figure 4.6(a), we find that the makespan in Figure 4.6(b) is larger than that in Figure 4.6(a). This is because the number of workers for task execution in the real cluster is larger than that of Amazon EC2. Also the real cluster is a local cluster, and it needs less time for transmission (e.g., data transmission).

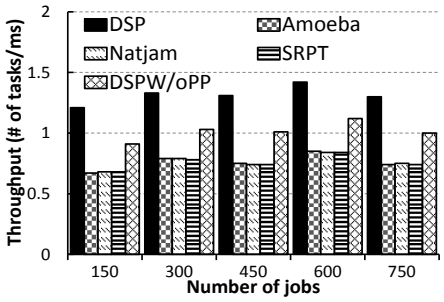
4.4.2 Effectiveness of Dependency Consideration in Preemption

Figure 4.7(a) shows the relationship between the number of disorders (execution order inconsistent with task dependency relation) and the number of jobs. We see that the number of disorders for DSP is always 0 and the result follows DSP<Natjam≈Amoeba<SRPT. In Figure 4.7(a), we find that the number of disorders for DSP is always 0 and the result follows DSP<Natjam≈Amoeba<SRPT. The reason behind this is DSP considers the dependencies among tasks when it schedules tasks, however the other methods do not substantially consider the dependencies among tasks when they schedule tasks. We also find the number of disorders increases as the number of jobs increases because more jobs increase the likelihood of disorder occurrences. In order to show the advantages of probabilistic based preemption, we also evaluated the performance of DSPW/oPP, a variant of DSP in which the PP is not considered. Figure 4.7(b) shows the relationship between the throughput measured by tasks/ms and the number of jobs. We see that the throughput follows SRPT<Amoeba≈Natjam<DSPW/oPP<DSP. This is because (1) DSP considers deadline constraints for job completion time; (2) DSP considers the dependencies among tasks and schedules tasks are independent of each other to different workers (processors) to run tasks in parallel so that it reduces the completion time and thus increases throughput. Also, DSP judiciously utilizes task dependency information to determine task priority, and assigns higher priority to tasks that have more dependent tasks, which helps increase the throughput; (3)Unlike the other methods for preemption, DSP uses the PP to reduce the time overhead caused by preemption and thereby increase the throughput, However, all three other methods do not substantially consider dependency in scheduling or preemption, and the tasks that depend on other tasks need to wait even for other tasks to finish, so they produce lower throughput than DSP. Throughput is affected by the remaining time. Amoeba only considers the remaining time to choose tasks, so it has relatively higher throughput. Natjam also considers task remaining time to choose task for eviction. Moreover, it takes into account job deadlines for scheduling tasks. Thus it has relatively higher throughput. SRPT considers task waiting time and remaining time for scheduling, and it gives some weight to waiting time, so its throughput is not so high as Amoeba or Natjam.

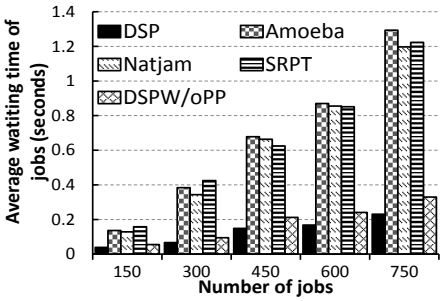
Figure 4.7(c) reveals the relationship between the average waiting time of jobs and the number of jobs. We see that the average waiting time of jobs approximately follows DSP<DSPW/oPP< Natjam≈SRPT<Amoeba. This is because DSP utilizes the task dependency information, which



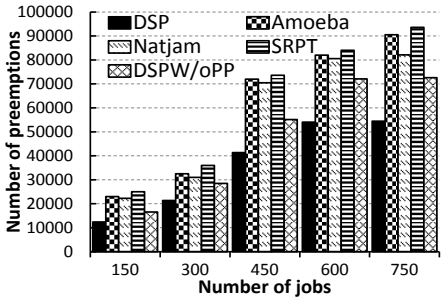
(a) Relationship between the number of disorders and the number of tasks



(b) Relationship between throughput and the number of tasks



(c) Relationship between jobs' average waiting time and the number of tasks



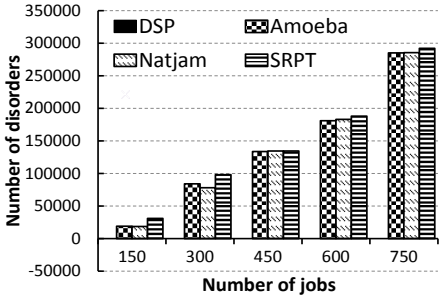
(d) Relationship between overhead and the number of tasks

Figure 4.7: Performance of various evaluation metrics versus the number of tasks across different methods on a real cluster.

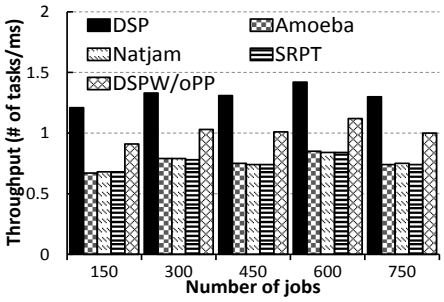
can increase task waiting time, to determine task priority for execution. Also, DSP considers task waiting time for determining task priority, which reduces the waiting time of low priority tasks and also can avoid starvation. In addition, DSP uses a probabilistic based approach for preemption (ignored in DSPW/oPP), which can reduce the time overhead caused by preemption and thus reduce the waiting time of tasks. Therefore DSP has the shortest average waiting time, and DSPW/oPP comes second. The average waiting times of jobs of Natjam and SRPT are nearly the same, but relatively shorter than that of Amoeba. This is because Natjam considers job deadlines and it leverages smart eviction policies to reduce the average waiting time of jobs. SRPT considers task waiting time to determine task priority for preemption, which can reduce the average waiting time of jobs. However, Amoeba neither considers task waiting time for determining task priority nor has deadline constraints for jobs, thus it has relative longer average waiting time of jobs. Figure 4.7(d) shows the relationship between overhead and the number of jobs for scheduling. In Figure 4.7(d), we use the number of preemptions to represent the overhead because the overhead is mainly

caused by preemption (e.g., context switch). From Figure 4.7(d), we observe that the overhead follows DSP<DSPW/oPP<Natjam<Amoeba<SRPT. This is because (1) DSP utilizes task dependency information to determine task priority and chooses tasks for running, which can decrease the probability of preemption because the running task could be preempted only if the waiting task does not depend on the running task; (2) DSP allows preemptions only for some top priority tasks which is neglected in DSPW/oPP. However, Natjam, Amoeba and SRPT do not use a probabilistic based strategy for preemption, which increases the number of preemptions. Thus, Natjam, Amoeba and SRPT have relative higher overhead than DSP and DSPW/oPP. Also Natjam supports preemption for only research jobs rather than production jobs. Thus it has relatively lower overhead caused by preemption than Amoeba and SRPT. Unlike all the other methods, SRPT does not use a checkpoint technique to reduce the overhead caused by preemption. Therefore, SRPT has the highest overhead among all the methods.

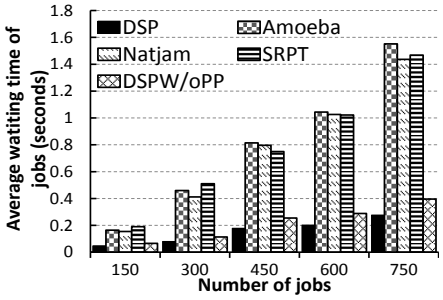
To further verify the performance of DSP, we also compared DSP with other methods on the metrics of the overhead, the number of disorders, the throughput and the average waiting time of jobs on Amazon EC2. Figure 4.8(a) shows the relationship between the number of disorders for executing tasks and the number of jobs on Amazon EC2. In Figure 4.8(a), we also find the number of disorders increases as the number of jobs increases, and the number of disorders follows DSP<Natjam≈Amoeba<SRPT due to the same reasons explained in Figure 4.7(a). Figure 4.8(b) shows the relationship between the throughput (# of tasks/ms) and the number of jobs on Amazon EC2. In Figure 4.8(b), we also see the throughput follows SRPT<Amoeba≈Natjam<DSPW/oPP< DSP. The reasons are the same as that explained in Figure 4.7(b). Figure 4.8(c) shows the relationship between the average waiting time of jobs and the number of jobs on Amazon EC2. From Figure 4.8(c), we also find the average waiting time of jobs roughly follows DSP<DSPW/oPP<Natjam≈ SRPT<Amoeba due to the same reasons explained in Figure 4.7(c). Comparing Figure 4.8(c) with Figure 4.7(c), we find the average waiting time of jobs in Figure 4.8(c) is longer than that in Figure 4.7(c). The reason is that the number of worker machines in the real cluster environment is larger than that in Amazon EC2, and tasks have more chances to find idle worker machines. Thus the average waiting time of jobs in Figure 4.8(c) is relatively longer than that in Figure 4.7(c). Figure 4.8(d) shows the relationship between the overhead and the number of jobs for scheduling on Amazon EC2. Similar to Figure 4.7(d), we also see the overhead increases as the number of jobs increases, and the overhead of these methods follows: DSP<DSPW/oPP<Natjam<Amoeba<SRPT



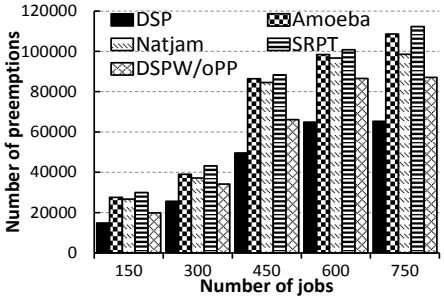
(a) Relationship between the number of disorders and the number of tasks



(b) Relationship between throughput and the number of tasks



(c) Relationship between jobs' average waiting time and the number of tasks



(d) Relationship between overhead and the number of tasks

Figure 4.8: Performance of various evaluation metrics versus the number of tasks across different methods on Amazon EC2.

due to the same reasons explained in Figure 4.7(d). Comparing Figure 4.8(d) to Figure 4.7(d), we find the overhead in Figure 4.8(d) is relatively higher than that in Figure 4.7(d). This is because the average number of tasks assigned to a worker in the real cluster environment is less than that of Amazon EC2 due to less workers in Amazon EC2, and preemption is more likely to occur for workers with more tasks.

4.5 Conclusion

This chapter presents a dependency-aware scheduling and preemption strategy (DSP) that provides high throughput with less overhead for parallel jobs. Specifically, the chapter first introduces the system model of DSP in which a makespan minimization model is illustrated. Next, it describes how to determine task priority with the consideration of dependency (i.e., the dependency-considered task priority determination strategy). Then, the chapter presents the probabilistic based preemption approach. Finally, the chapter describes the experimental results based on a real cluster and Amazon

EC2, which show that DSP achieves high throughput, low overhead and job satisfaction compared to existing scheduling and preemption strategies.

Chapter 5

Related Work

In this chapter, we review the related work on each of the proposed research problems in this dissertation. The related work is split into three main categories. First, we give an overview of existing works on handling non-correlated or correlated machine failures and existing works aiming at improving data availability for distributed storage system. Second, we review existing works on resource allocation (or resource provisioning) aiming at increasing resource utilization. Third, we go over existing works on job (task) scheduling with the objective of increasing throughput.

5.1 Machine Failures Handling and Data Availability Enhancement

Many methods have been proposed to handle non-correlated or correlated machine failures. Zhong *et al.* [139] assumed independent machines failures, and proposed a model that achieves high expected service availability under given space constraint, object request popularities, and object sizes. However, this model does not consider correlated machine failures hence cannot handle such failures, especially in the scenario of severe machine failures correlations. Nath *et al.* [86] identified a set of design principles that system builders can use to tolerate correlated failures. Cidon *et al.* proposed Copyset Replication [42] and Tiered Replication [41] to reduce the frequency of data loss caused by correlated machine failures by limiting the replica nodes of many chunks to a single copyset. These methods for correlated machine failures do not consider data popularity to

minimize data loss probability in correlated and non-correlated machine failures. Unlike Copyset Replication and Tiered Replication, our proposed MRR reduces probability of data loss caused by both correlated and non-correlated machine failures with considering data popularity, and it derives diverse replication degrees for data objects with different popularities and thus increases the overall data object availability by creating more replicas for popular data objects and less for unpopular data objects. Moreover, MRR replicates data objects with the consideration of reducing consistence maintenance cost and storage cost, which is critical for cloud providers to maximize the revenue. Ford *et al.* [53] analyzed different failure loss scenarios on GFS clusters, and proposed geo-replication as an effective technique to prevent data loss under large scale concurrent node failures, however they did not provide a specific strategy for reducing data loss caused by both correlated and non-correlated machine failures while minimizing cost (e.g., consistency maintenance cost and storage cost) caused by replication. Thomson *et al.* [117] presented CalvinFS, a scalable distributed file system, which horizontally partitions and replicates file system metadata across a shared-nothing cluster of servers. However, CalvinFS cannot handle both correlated and non-correlated machine failures while minimizing cost (e.g., consistency maintenance cost and storage cost) caused by replication. The above previously proposed methods cannot handle both correlated machine failures and non-correlated machine failures while utilizing the limited resource to increase data availability. They neglect data popularity existing in current cloud storage system and thus cannot fully utilize the resource to increase data availability. Also, they do not concurrently consider minimizing the consistency maintenance cost and storage cost caused by replication, which is important for cloud providers to maximize the revenue.

High availability has been accepted as an explicit requirement for distributed storage systems. There is a large body of work on enhancing data availability for distributed storage systems. Duminuco *et al.* [50] proposed a framework for data replication based on an ongoing estimation of the peer failure behavior in a distributed storage system. They tried to estimate the availability of a machine in a distributed storage system in order to replicate its data and thereby enhance data availability. Abu-Libdeh *et al.* [13] presented a replication protocol for datacenter services in order to provide strong consistency and high availability. A protocol running within shards (a collection of nodes) ensures linearizable consistency, while the shards interact in order to improve availability. Rajagopalan *et al.* [97] proposed Pico Replication (PR), a system-level high availability framework for middleboxes that operates on individual flows and independently and transparently replicate

flow-specific state through continuous live migration and output buffering. PR leverages OpenFlow to provide near instant flow-level failure recovery, by dynamically rerouting a flow's packets to its replication target. Ko *et al.* [67] minimized the effect of run-time server failures on the availability of intermediate data, and designed a new storage system ISS (Intermediate Storage System) that treats intermediate data in dataflow programs as a first-class citizen in order to tolerate failures. Litwin *et al.* [74] proposed a new high-availability Scalable Distributed Data Structure (SDDS), and it uses the concept of record grouping and the Reed Salomon codes to provide scalable, distributed and high-availability files, badly needed by modern application. Bonvin *et al.* [28] proposed a self-managed key-value store that dynamically allocates the resources of a data cloud to several applications in a cost-efficient and fair manner. Their approach offers and dynamically maintains multiple differentiated availability guarantees to each different application despite of failures. Ford *et al.* [53] characterized the availability properties of cloud storage systems based on an extensive one year study of Google's main storage infrastructure and presented statistical models that enable further insight into the impact of multiple design choices, such as data placement and replication strategies. Zhang *et al.* [137] presented Mojim to provide the reliability and availability in large-scale storage systems. Mojim uses a two-tier architecture in which the primary tier contains a mirrored pair of nodes and the secondary tier contains the secondary backup nodes with weakly consistent copies of data. Yu *et al.* [132] used a fixed number of replicas for every data object, and showed that the assignment of object replicas to machines plays a dramatic role in the availability of multi-object operations. Bhagwan *et al.* [26] studied the estimation of machine availability in a distributed system and its utilization in guiding replication degree and replica placement. However, most of these previous works neglect data object popularities when determining the number of replicas for each object, thus cannot satisfy the demands of popular data objects or fully utilize the limited resource. Although the works in [18,139] consider object popularities, they do not give a solution for handling correlated machine failures, which is a key issue in achieving high availability in today's cloud datacenters [86]. In addition, they do not consider different storage medium prices (for reducing storage cost) or geographic distance in selecting nodes to store replicas, both of which affect the total cost of the storage systems.

Motivated by the problems in the existing works, we propose MRR that can effectively handle both correlated and non-correlated machine failures and also considers the factors indicated in the introduction section to maximize data availability and reduce the consistency maintenance

cost and storage cost.

5.2 Resource Allocation for High Resource Utilization

Resource provisioning with high resource utilization becomes increasingly important in computing and communication systems in recent years. Many works focusing on resource provisioning try to improve the resource utilization. However their approaches are either based on reservation or demand-based resource allocation, which cannot fully utilize the resource all the time. Shen *et al.* [108] presented CloudScale, a system that automates fine-grained elastic resource scaling for multi-tenant cloud computing infrastructures. CloudScale uses online resource demand prediction and prediction error handling to achieve adaptive resource allocation, that is, it uses a hybrid approach that employs signature-driven and state-driven prediction algorithms to achieve both high accuracy and low overhead. Specifically, the model uses a fast Fourier transform (FFT) to identify repeating patterns (called signatures) to estimate future resource demands (if no repeating patterns is found, it adopts a discrete-time Markov chain to predict the resource demand in the near future), then the prediction error correction module performs online adaptive padding that adds a dynamically determined cushion value to the predicted resource demand to avoid under-estimation errors. However, the method in the work [108] is demand-based resource allocation which cannot fully utilize the resource because users typically get a small ceiling of resource from the resource providers. Also, it uses either the maximum or the 80% of the high-frequency spectrum as padding, and it raises the cap by the ratio α ($\alpha > 1$) to correct the underestimate error without decreasing the cap when it is much higher than the actual demand. Thus it cannot fully utilize the resource. In addition, it performs long-term conflict prediction, but the prediction is not accurate when no repeated pattern can be found in the training data because it relies on multi-step Markov prediction which has limited accuracy. Urgaonkar *et al.* [119] proposed a method for dynamically provisioning CPU and network resources in shared hosting platforms. To accurately estimate an application's resource, they presented techniques to profile applications on dedicated nodes and used the profiles to guide the placement of application components onto shared nodes. Also, they presented techniques to overbook cluster resources in a controlled fashion such that the platform can provide performance guarantees to applications even when overbooked. However, it involves additional adaption cost such as data migration and buffer pool warmup. Curino *et al.* [43] proposed a reservation-

based scheduling, which consists of two main processes: reservation and planning. The process of reservation determines a job's resource needs and temporal requirements, and translates the job's completion service level agreements (SLA) into a service level objective (SLO) over predictable resource allocations, which is done ahead of job's execution. The process of planning involves constructing a temporal assignment of cluster resources to jobs such that each job's RDL (reservation definition language which provides a uniform and abstract representation of all the jobs's needs) expression is satisfied. However, resource reservation can lead to resource underutilization as users do not necessarily understand complex physical resource requirements of jobs [48]. Delimitrou *et al.* [48] presented Quasar, a cluster management system that increases resource utilization while providing consistently high application performance. First, Quasar allows users to communicate the performance constraints of the application in terms of throughput and/or latency, depending on the application type, and it determines the amount of the available resources needed to meet performance constraints based on users' expression of performance constraints. Second, Quasar relies on classification techniques to quickly and accurately determine the impact of the amount of resources (scale-out and scale-up), type of resources, and interference on performance for each workload. Third, Quasar performs resource allocation and assignment jointly. The classification results are used to determine the right amount and specific set of resources assigned to the workload. However, Quasar requires users to communicate performance constraints for each workload and it is still on-demand based resource allocation, and thus the resource cannot be fully utilized all the time because of the temporarily unused resource [34]. Shanmuganathan *et al.* [103] proposed two algorithms DBS and BPX to dynamically allocate the capacity among a tenant's VMs based on their dynamic demand, shares, reservation, and limit settings. Specifically the tenant transparently multiplex this purchased capacity dynamically among its VMs. The two algorithms DBS and BPX periodically re-distribute the purchased capacity among the tenant's VMs based on their demand, and other tenant-specified controls such as reservations, limits and priority (or shares). However, the approach is a demand-based resource allocation in which the capacity will not be fully utilized all the time due to the allocated unused resource. However, the above works do not focus on reallocating the allocated unused resources to increase the resource utilization.

To increase resource utilization, some works [34, 83] and the product [3] provide methods of offering resources opportunistically. Marshall *et al.* [83] presented reusing unused cloud resources by offering leases in an opportunistic and preemptible way with no SLOs. Although the method

proposed in the work [83] can increase the cloud utilization to some degree, they do not ensure SLOs, which affects the QoS because the resource can be preempted at any time. Also they do not provide a method to recover from SLO violations when SLO violations occur. Amazon EC2 Spot Instances [3] is a product that offers opportunistic resources. The approach can increase the resource utilization, but the offered resources have no SLOs, which affects the QoS when the allocated resource for processing jobs is preempted. Recently, Carvalha *et al.* [34] showed how to provide strong, long-term availability SLOs for a portion of the unused resources. The method in [34] uses time series forecasting with the assumption that the resource usage patterns exist in training data to predict the unused used resource for long-term service jobs. The SLOs are based on forecasts of how many resources will remain unused during multi-month periods so users can do capacity planning for their long-running services. However, this method is not suitable for processing short-lived jobs because such jobs usually do not exhibit certain resource utilization patterns. Also, it fails to consider fluctuations of unused resource caused by time-varying resource demands of short-lived jobs. In addition, the method may result in resource fragmentation and lead to low resource utilization because they neglect jobs' resource intensity in multi-resource allocation and may allocate much more resources to the jobs.

Unlike previous works, we propose a deep learning-based cooperative opportunistic resource provisioning (CORP) scheme. CORP first predicts the amount of allocated but unused resource using the deep learning technique, in which the accuracy does not rely on the existence of resource utilization patterns of short-lived jobs. To well meet the requirement of time-varying job resource demands, CORP additionally considers the fluctuations of unused resource and uses HMM model to correct prediction errors. Also, CORP leverages complementarity of jobs' requirements on different resource types and packs jobs with complementary resource requirements to VMs to reduce the resource fragmentation and further increase the resource utilization. Thus our proposed method CORP can fully utilize the resource while reducing SLO violation rate.

5.3 Scheduling for High Throughput

Many methods have been proposed for job scheduling or task scheduling with the objective of maximizing throughput. The works [23, 88–90, 95, 126, 135] focus on maximizing the throughput in scheduling. Previous scheduling algorithms to increase throughput can be roughly classified

into two categories. The first category [88–90] focuses on scheduling short (sub-second) tasks for high throughput. The second category [23, 95, 126, 135] focuses on scheduling long-running batch jobs/tasks. Ousterhout *et al.* [89] proposed a decentralized load balancing approach (called batching sampling) that provides near-optimal performance for scheduling highly parallel jobs. Batching sampling generalizes Mitzenmacher's "power of two random choices" load balancing technique in which an incoming task is assigned to the least-loaded of two randomly chosen slaves. Batch sampling does not relying on shared state, scheduling can be performed by a large number of distributed schedulers that place tasks in parallel, the distributed architecture increases the throughput. Xin *et al.* [129] presented Shark which leverages a novel distributed memory abstraction to provide a unified engine that can run SQL queries and sophisticated analytics functions (e.g., iterative machine learning) at scale, and efficiently recovers from failures mid-query. Ousterhout *et al.* [88] presented an architecture that supports tiny tasks. It breaks data-parallel jobs in compute clusters into tiny tasks so that each can complete in hundreds of milliseconds. However, all these works for short task scheduling neglect utilizing task dependency in scheduling to increase throughput. Zaharia *et al.* [135] proposed a delay scheduling: when the job that should be scheduled next according to fairness cannot launch a local disk, it waits for a small amount of time, letting other jobs launch tasks instead. The work [135] shows delay scheduling achieves nearly optimal data locality in a variety of workloads and can increase throughput by up to 2x while preserving fairness. To increase the throughput, Raicu *et al.* [95] proposed a "data diffusion" approach that acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. Wang *et al.* [126] struck the right balance between data-locality and load-balancing to maximize throughput. They presented a new queueing architecture and proposed a map task scheduling algorithm constituted by the *Join the Shortest Queue policy* together with the *MaxWeight policy* [116]. Bar-Noy *et al.* [23] tried to find a nonpreemptive schedule with batching that maximizes the throughput of the scheduled jobs. Specifically, it partitions jobs into a certain number of families (i.e., each family is associated a processing time), and batches and executes the same family jobs together (on the same machine) in the same time that it takes to execute a single job from that family. However, all these works for batch jobs/tasks scheduling neglect dependency in scheduling to increase throughput. Although Grandl *et al.* [57] proposed a multi-resource cluster scheduler Tetris which considers task dependency and packs tasks to machines based on the alignment score, it neglects job deadline constraints which can delay jobs' completion time and thus decrease the throughput. Also,

by simply considering task dependency, Tetris cannot fully utilize task dependency information to increase the throughput by judiciously determining tasks' execution order so that more tasks have chance to start executing after a chosen task finishes execution. Unlike previous works, we consider task dependency and assign tasks that are of independent of each other to different workers (or processors) so that the independent tasks can run in parallel and thus increase throughput.

There is a large body of preemption works [19, 22, 40, 59] aiming at reducing the waiting time of high priority tasks. Based on the number of preemptions and context switching overhead, these existing works can be divided into two categories. The first category [19, 22, 40] tries to satisfy high priority tasks based on the remaining time, resource consumption, waiting time, etc. The second category [59] aims at maximizing the throughput with less number of preemptions. Ananthanarayanan *et al.* [19] proposed Amoeba which chooses the tasks that consume the most resources to be preempted. Amoeba uses a checkpoint/restart mechanism to dynamically change the number of slots assigned to a task. Amoeba finds the safe point to evict a task, hence it does not save intermediate states. Cho *et al.* [40] proposed Natjam which uses an on-demand checkpointing technique that saves the state of a task when it is preempted, so that it can resume where it left off when resources become available. Natjam classifies jobs into production jobs and research jobs, and it assigns higher priority to production jobs and lower priority to research jobs. Natjam offers two types of evictions: job eviction and task eviction. Balasubramanian *et al.* [22] proposed decentralized preemptive scheduling strategies aiming at maximizing total throughput and minimizing average waiting time. It first determines job priority based on the remaining time and the waiting time, and then it performs preemption based on the determined priorities. However, all these works suffer from the overhead of context switching, which decreases the throughput. Harchol-Balter *et al.* [59] proposed size-based scheduling and preemption. It gives preference (priority) to requests for small files or requests with short remaining file size so that the throughput can be increased. However, lowest priority tasks may experience long waiting time in this method, which can decrease the throughput. Unlike previous works that consider only one or two of the factors, we consider the combination of the factors (e.g., the number of dependent tasks, task's remaining time, task's waiting time) to determine task priority, which is more fair and can avoid the long waiting time of low priority tasks.

In our preemptive scheduling approach, not only we consider the dependency relations among different tasks for scheduling (task assignment) and preemption, but also we consider reducing

the overhead of extensive number of preemptions for maximizing throughput.

Chapter 6

Conclusion

Cloud computing is an emerging technology in distributed computing, and it has proved to be an effective infrastructure to provide services to users. Cloud is developing day by day and faces many challenges: building cost-effective data management system that can ensure high data availability while maintaining consistency [15]; efficient resource allocation providing high resource utilization and high SLO availability [75, 123]; scheduling jobs (tasks) with high throughput [14]. To address these challenges, in this dissertation, we study how to manage data and improve data availability while reducing cost (i.e., consistency maintenance cost and storage cost); how to efficiently manage the resource for processing jobs and increase the resource utilization with high SLO availability; how to design an efficient scheduling algorithm which provides high throughput, low overhead while satisfying the demands on completion time of jobs. We summarize the works in this dissertation below.

Previous replication schemes for cloud storage systems consider correlated machine failures or non-correlated machine failures to reduce data loss. However, no previous methods can effectively handle both types of machine failures, which co-exist in cloud storage systems, and consider data popularity and fully utilize the limited resource to maximize data availability simultaneously. Also, although data replicas bring about additional cost in storage and consistency maintenance, few previous schemes aim to simultaneously minimize both costs by considering the replication degrees, different storage mediums and the geographic distances of the replica nodes. In this dissertation, in order to increase data availability and reduce cost caused by replication, we formulate an optimization problem that determines the replication degree of each data object so that the request failure

probability, consistency maintenance cost and storage cost are minimized in a cloud storage in both correlated and non-correlated machine failures. Based on the problem solution, we propose the MRR scheme that assigns the chunk replicas of data objects to the nodes to handle the aforementioned problems for the objective. Our numerical analysis and real-word experiments on Amazon S3 show that MRR outperforms other replication schemes in different performance metrics.

In order to increase the resource utilization and reduce SLO violation rate, we propose CORP for short-lived jobs, which offers the temporarily-unused resource in an opportunistic manner. CORP is different from previous works in three aspects. First, using the deep learning technique, it can more accurately predict the amount of allocated and unused resources of short-lived jobs, which do not have resource usage patterns. Second, it additionally considers the fluctuations of unused resource caused by time-varying resource demands of jobs to correct the prediction. Third, it leverages complementarity of jobs' requirements on different resource types and packs jobs with complementary requirements on resources to the same VM to reduce resource fragmentation and further increase the resource utilization. Experimental results based on a real cluster and Amazon EC2 show that our method achieves high resource utilization and provides high SLO guarantee.

To provide high throughput with less overhead for parallel jobs, we present DSP, which takes into account task dependency in task assignment and preemption, and further reduces the job response time by running tasks that are independent of each other in parallel. To satisfy high priority jobs without incurring much overhead, we judiciously utilize task dependency information and jointly consider task remaining time and waiting time for determining task priority, and we propose a probabilistic based preemption method which can reduce the waiting time of high priority task and the time overhead while avoiding starvation for low-priority task. We compare our method with the existing methods under different scenarios using a large-scale real cluster and Amazon EC2 cloud service. Our preliminary results shows that DSP outperforms the existing methods under both the real cluster and Amazon EC2 cloud service.

In the future, we will further consider data update frequency to reduce consistency maintenance cost, and we will consider the effects of node joining, node leaving, and the influence of changing network connections. Also we will consider energy consumption of machines and design an optimal replication scheme to improve data availability while saving power. For resource allocation, we will consider using the greedy approach for deep learning to reduce the computation overhead caused by the deep neural network. Also, we will additionally consider the heterogeneity of jobs

(i.e., short jobs and long jobs) existing in current cloud systems. In order to increase the resource utilization, on the one hand, we will further consider jobs' complementary requirements on different resource types (in spatial space), and on the other hand, we will consider jobs' execution time and classify jobs into short jobs and long jobs, and pack the complementary jobs with the same job type and allocate the resource to them. In addition, we will use a hybrid resource allocation strategy and provide SLO availability customization for different job types. Specifically, we will use the opportunistic-based resource allocation (ORA) for short jobs with relatively lower SLO availability guarantee for achieving high resource utilization, and use the demand-based resource allocation (DRA) for long jobs with higher SLO availability guarantee for achieving high resource utilization and low SLO violation rate simultaneously. For scheduling, we will aim to handle scheduling tasks with partial dependency, worker failures in job (task) scheduling and make our DSP fully distributed to increase its scalability. Finally, we plan to use different workloads and real-world experiment to fully test performance of our methods under various scenarios and make our preliminary system design more mature.

Bibliography

[1] http://www.inf.ed.ac.uk/teaching/courses/ds/slides/systems-and-models.pdf [accessed in May 2015].

[2] Amazon EC2. http://aws.amazon.com/ec2 [accessed in May 2015].

[3] Amazon EC2 instance purchasing options. https://aws.amazon.com/ec2/purchasing-options [accessed in May. 2015].

[4] Capacity scheduler. http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html [accessed in May 2015].

[5] Cplex linear program solver. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/ [accessed in May 2015].

[6] Faqs. https://aws.amazon.com/s3/faqs/ [accessed in Jun. 2016].

[7] Google trace. https://code.google.com/p/googleclusterdata/ [accessed in May 2015].

[8] Ipopt. https://projects.coin-or.org/Ipopt [accessed in May. 2015].

[9] Palmetto cluster. http://citi.clemson.edu/palmetto/ [accessed in May 2015].

[10] Spec 2007, standard performance evaluation corporation. http://www.spec.org/ [accessed in May 2015].

[11] Intelligent block placement policy to decrease probability of data loss. https://issues.apache.org/jira/browse/HDFS-1094 [accessed in May 2015].

[12] M. Abdeyazdan, S. Parsa, and A. Rahmani. Task graph pre-scheduling, using nash equilibrium in game theory. *Journal of Supercomputing*, 64(1):177–203, 2013.

[13] H. Abu-Libdeh, R. Renesse, and Y. Vigfusson. Leveraging sharding in the design of scalable replication protocols. In *Proc. of SoCC*, 2013.

[14] A. Agarwal and S. Jain. Efficient optimal algorithm of task scheduling in cloud computing environment. *International Journal of Computer Trends and Technology (IJCTT)*, 9(7):344–349, 2014.

[15] D. Agrawal, A. E. Abbadi, S. Antony, and S. Das. Data management challenges in cloud computing infrastructures. In *Proc. of DNIS*, pages 1–10, 2010.

[16] M. Alicherry and T. Lakshman. Optimizing data access latencies in cloud systems by intelligent virtual machine placement. In *Proc. of INFOCOM*, pages 647–655, 2013.

[17] M. Alizadeh, A. Javanmard, S. Chuang, S. Iyer, and Y. Lu. Versatile refresh: Low complexity refresh scheduling for high-throughput multi-banked edram. In *Proc. of SIGMETRICS*, 2012.

[18] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Copingwith skewed content popularity inmapreduce clusters. In *Proc. of EuroSys*, Salzburg, April 2011.

[19] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of SoCC*, 2012.

[20] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI*, 2010.

[21] F. André, A. Kermarrec, E. Merrer, N. Scouarnec, G. Straub, and A. Kempen. Archiving cold data in warehouses with clustered network coding. In *Proc. of EuroSys*, 2014.

[22] A. Balasubramanian, A. Sussman, and N. Sadeh. Decentralized preemptive scheduling across heterogeneous multi-core grid resources. In *Proc. of JSSPP*, 2013.

[23] A. Bar-Noy, S. Guha, Y. Katz, J. Naor, B. Schieber, and H. Shachnai. Throughput maximization of real-time scheduling with batching. *ACM Trans. on Algorithms*, 5(2):1–17, 2009.

[24] M. Bazaraa, H. Sherali, and C. Shetty. Nonlinear programming: Theory and algorithms. *Wiley Interscience*, 2006.

[25] Y. Bengio. Learning deep architectures for ai. Technical report, Universite de Montreal, 2007.

[26] R. Bhagwan, K. Tati, Y. Cheng, s. Savage, and G. Voelker. Total recall: System support for automated availability management. In *Proc. of NSDI*, pages 337–350, San Francisco, CA, March 2004.

[27] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. of SIGMETRICS*, 2000.

[28] N. Bonvin, T. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *Proc. of SoCC*, Indianapolis, 2010.

[29] N. Bonvin, T. G. Papaioannou, and A. Aberer. Autonomic sla-driven provisioning for cloud applications. In *Proc. of CCGrid*, pages 434–443, 2011.

[30] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *Proc. of SIGMOD*, 2011.

[31] L. Bottou. Stochastic gradient learning in neural networks. In *Proc. of Neuro-Nimes*, Nimes, 1991.

[32] E. Boutin, J. Ekanayake, W. Lin, B. Shi, and J. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proc. of USENIX OSDI*, 2014.

[33] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, and K. M. L. Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proc. of SOSP*, 2011.

[34] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term slos for reclaimed cloud computing resources. In *Proc. of SoCC*, Seattle, 2014.

[35] R. Chansler. Data availability and durability with the hadoop distributed file system. *The USENIX Magazine*, 2012.

[36] C. Chatfield. *The analysis of time series.* Texts in Statistical Science, Chapman & Hall, sixth edition, 2004.

[37] K. Chen and H. Shen. Dsearching: Distributed searching of mobile nodes in dtns with floating mobility information. In *Proc. of INFOCOM*, 2014.

[38] S. Chen, L. Xiao, and X. Zhang. Adaptive and virtual reconfigurations for effective dynamic job scheduling in cluster systems. In *Proc. of ICDCS*, 2002.

[39] D. Chicco, P. Sadowski, and P. Sadowski. Deep autoencoder neural networks for gene ontology annotation predictions. In *ACM BCB*, 2014.

[40] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proc. of SoCC*, Santa Clara, October 2013.

[41] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proc. of ATC*, pages 31–43, 2015.

[42] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proc. of Usenix ATC*, 2013.

[43] C. Curinom, D. E. Difallahu, C. Douglasm, S. Krishnanm, R. Ramakrishnanm, and S. Raom. Reservation-based scheduling: If you're late don't blame us! In *Proc. of SoCC*, Seattle, Nov. 2014.

[44] J. Dean. Evolution and future directions of large-scale storage and computation systems at google. In *Proc. of SoCC*, page 1, 2010.

[45] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[46] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

[47] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of ATC*, 2015.

[48] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of ASPLOS*, pages 127–143, Salt Lake City, March 2014.

[49] S. Di, Y. Robert, F. Vivien, D. Kondo, C. Wang, and F. Cappello. Optimization of cloud task processing with checkpoint-restart mechanism. In *Proc. of SC*, 2013.

[50] A. Duminuco, E. Biersack, and T. En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *Proc. of CoNEXT*, New York, December 2007.

[51] S. Dutta and A. Verma. Service deactivation aware placement and defragmentation in enterprise clouds. In *Proc. of CNSM*, pages 1–9, 2011.

[52] A. Feldmann, M. Kao, J. Sgalli, and S. Teng. Optimal online scheduling of parallel jobs with dependencies. In *Proc. of STOC*, 1993.

[53] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of OSDI*, Berkeley, CA, USA, 2010.

[54] W. Gao and G. Cao. Fine-grained mobility characterization: Steady and transient state behaviors. In *Proc.of MOBIHOC*, Chicago, 2010.

[55] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of SOSP*, pages 29–43, 2003.

[56] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Proc. of CNSM*, 2010.

[57] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proc. of SIGCOMM*, Chicago, 2014.

[58] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, 2005.

[59] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. on Computer Systems*, 21(2):207–233, 2003.

[60] F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw-Hill Higher Education, 2005.

[61] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: characterization of the mapreduce-based data analysis. In *Proc. of ICDEW*, 2010.

[62] M. Iverson, F. ´Ozg´uner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Trans. Comput*, 48(12):1374–1379, 1999.

[63] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. of ACM SIGCOMM*, 2015.

[64] P. Kaski and P. Östergård. There exists no (15, 5, 4) rbibd. *Journal of Combinatorial Designs*, 9:227–232, 2001.

[65] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proc. of ISCA*, pages 350–361, Saint-Malo, June 2010.

[66] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proc. of USENIX FAST*, 2012.

[67] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proc. of SoCC*, Indianapolis, 2010.

[68] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[69] N. D. Lane and P. Georgiev. Can deep learning revolutionize mobile sensing? In *Proc. of HotMobile*, Santa Fe, 2015.

[70] E. Lawler and J. Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *JACM*, 25(4):612–619, 1978.

[71] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng. On optimization methods for deep learning. In *Proc. of ICML*, 2011.

[72] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[73] J. Lee and K. Shin. Preempt a job or not in edf scheduling of uniprocessor systems. *IEEE Transactions on Computers (TC)*, 63(5):1197–1205, May 2014.

[74] W. Litwin and T. Schwarz. Lh*rs: A high-availability scalable distributed data structure using reed solomon codes. In *Proc. of ACM SIGMOD*, May 2000.

[75] J. Liu, H. Shen, and L. Chen. Corp: Cooperative opportunistic resource provisioning for short-lived jobs in cloud systems. In *Proc. of the 2016 IEEE International Conference on Cluster Computing (Cluster)*, September 2016.

[76] J. Liu, H. Shen, and L. Yu. Question quality analysis and prediction in community question answering services with coupled mutual reinforcement. *IEEE Trans. on Services Computing (TSC)*, PP(99):1–14, 2015.

[77] J. Liu, H. Shen, and X. Zhang. A survey of mobile crowdsensing techniques: A critical component for the internet of things. In *Proc. of the 25th International Conference on Computer Communication and Networks (ICCCN) Workshop on ContextQoS*, Waikoloa, Hawaii, USA, 2016.

[78] J. Liu, L. Yu, H. Shen, Y. He, and J. Hallstrom. Characterizing data deliverability of greedy routing in wireless sensor networks. In *Proc. of SECON*, Seattle, June 2015.

[79] J. Liu, L. Yu, H. Shen, Y. He, and J. Hallstrom. Characterizing data deliverability of greedy routing in wireless sensor networks. Technical Report TR-2015-04, Clemson University, 2015 http://shenh.people.clemson.edu/publishedPaper/Report/2015/WSNTechRep.pdf.

[80] Y. Lv, Y. Duan, W. Kang, Z. Li, and F. Wang. Traffic flow prediction with big data: A deep learning approach. *ITS*, 16(2):865–873, 2015.

[81] Y. Ma, T. Nandagopal, K. P. N. Puttaswamy, and S. Banerjee. An ensemble of replication and erasure codes for cloud file systems. In *Proc. of INFOCOM*, pages 1276–1284, 2013.

[82] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proc. of SC*, 2011.

[83] P. Marshall, K. Keahey, and T. Freeman. Improving utilization of infrastructure clouds. In *Proc. of IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 205–214, 2011.

[84] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proc. of IEEE/ACM ICAC*, pages 11–20, Washington, DC, 2010.

[85] N. Nakka, A. Choudhary, R. Klundt, M. Weston, and L. Ward. Detailed analysis of i/o traces for large scale applications. In *Proc. of HiPC*, 2009.

[86] S. Nath, H. Yu, P. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proc. of NSDI*, pages 225–238, San Jose, CA, May 2006.

[87] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proc. of SOSP*, pages 29–41, 2011.

[88] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Proc. of HotOS*, 2013.

[89] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Batch sampling: Low overhead scheduling for sub-second prallel jobs. University of California, Berkeley, 2012.

[90] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. of SOSP*, 2013.

[91] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng. Locality-aware dynamic vm reconfiguration on mapreduce clouds. In *Proc. of HPDC*, pages 27–36. ACM, 2012.

[92] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. de Supinski. Practical resource management in power-constrained, high performance computing. In *Proc. of HPDC*, 2015.

[93] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proc. of IEEE Cluster Computing*, 2008.

[94] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of IEEE*, 77(2):257–286, 1989.

[95] I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain. The quest for scalable support of data-intensive workloads in distributed systems. In *Proc. of HPDC*, 2009.

[96] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *Proc. of SC*, Reno, November 2007.

[97] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proc. of SoCC*, Santa Clara, 2013.

[98] V. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Valuepack:value-based scheduling framework for cpu-gpu clusters. In *Proc. of SC*, 2012.

[99] V. Rawat. Reducing failure probability of cloud storage services using multi-clouds. In *Proc. of CoRR*, 2013.

[100] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of SoCC*, San Jose, October 2012.

[101] J. J. S. Houghten, L. Thiel and C. Lam. There is no (46, 6, 1) block design*. *Journal of Combinatorial Designs*, 9(1):60–71, 2001.

[102] L. E. Schrage and L. W. Miller. The queue m/g/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1996.

[103] G. Shanmuganathan, A. Gulati, and P. Varman. Defragmenting the cloud using demand-based resource allocation. In *Proc. of SIGMETRICS*, pages 67–80, Pittsburgh, June 2013.

[104] U. Sharma, P. Shenoy, and S. Sahu. A flexible elastic control plane for private clouds. In *Proc. of CAC*, Miami, August 2013.

[105] H. Shen, Z. Li, Y. Lin, and J. Li. SocialTube: P2P-assisted Video Sharing in Online Social Networks. *TPDS*, 2014.

[106] H. Shen, Z. Li, J. Liu, and J. E. Grant. Knowledge sharing in the online social network of yahoo! answers and its implications. *IEEE Transactions on Computers (TC)*, 64(6):1715C1728, June 2015.

[107] H. Shen, J. Liu, K. Chen, J. Liu, and S. Moyer. Scps: A social-aware distributed cyber-physical human-centric search engine. *IEEE Transactions on Computers (TC)*, 64:518–532, 2015.

[108] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. of SoCC*, Cascais, Oct. 2011.

[109] W. Shi, L. Zhang, C. Wu, Z. Li, and F. Lau. An online auction framework for dynamic resource provisioning in cloud computing. In *Proc. of SIGMETRICS*, Austin, June 2014.

[110] A. L. Shimpi. The ssd anthology: Understanding ssds and new drives from ocz. February 2014.

[111] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, pages 1–10, 2010.

[112] A. Smith, W. Min, B. Mott, and J. Lester. Diagrammatic student models: Modeling student drawing performance with deep learning. In *Proc. of UMAP*, 2015.

[113] M. Stamp. A revealing introduction to hidden markov models. January 18, 2004, http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf.

[114] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proc. of NSDI*, 2004.

[115] T. Taskaya-Temizel and M. Casey. Conjiguration of neural networks for the analysis of seasonal time series. In *Proc. of ICAPR*, 2005.

[116] L. Tassiulas and A. Ephremides. Dynamic server allocation to parallel queues with randomly varying connectivity. *IEEE Trans. Inf. Theory*, 39:466–478, Mar 1993.

[117] A. Thomson and D. Abadi. Calvinfs: Consistent wan replication and scalable metadata management for distributed file systems. In *FAST'15*.

[118] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[119] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proc. of OSDI*, 2002.

[120] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proc. of VLDB*, pages 501–510, Roma, 2001.

[121] V. Vavilapalli, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. OMalley, and S. R. B. R. E. Baldeschwielerh. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of SoCC*, Santa Clara, October 2013.

[122] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *Proc. of OSDI*, 2014.

[123] V. Vinothina, R. Sridaran, and P. Ganapathi. A survey on resource allocation strategies in cloud computing. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 3(6):97–104, 2012.

[124] A. Wächter and L. Biegler. On the implementation of an interior-point filter linesearch algorithm for large-scale nonlinear programming. *Math. Program*, 106(1):25–57, 2006.

[125] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang. Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. *In Proc. of INFOCOM*, 2013.

[126] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang. A throughput optimal algorithm for map task scheduling in mapreduce with data locality. *SIGMETRICS Performance Evaluation Review*, 40(4):33–42, 2013.

[127] X. Wang, E. Perlman, R. Burns, T. Malik, T. Budavari, C. Meneveau, and A. Szalay. Jaws: Job-aware workload scheduling for the exploration of turbulence simulations. In *Proc. of SC*, 2010.

[128] M. Wittie, V. Pejovic, L. Deek, K. Almeroth, and B. Zhao. Exploiting locality of interest in online social networks. In *Proc. of CoNEXT*, Philadelphia, 2010.

[129] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proc. of SIGMOD*, New York, 2013.

[130] Y. Xu, P. Tang, and J. Liu. Resource scheduling algorithm based on multi-target balance in enterprise cloud storage system. *Journal of Theoretical and Applied Information Technology*, 48(3):1578–1583, 2013.

[131] G. You, S. Hwang, and N. Jain. Scalable load balancing in cluster storage systems. In *Proc. of Middleware*, 2011.

[132] H. Yu, P. Gibbons, and S. Nath. Availability of multi-object operations. In *Proc. of NSDI*, pages 211–224, 2006.

[133] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proc. of SOSP*, pages 29–42, 2001.

[134] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: Deep learning in android malware detection. In *Proc. of ACM SIGCOMM*, pages 371–372, 2014.

[135] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*, 2010.

[136] J. Zhang, G. Tian, Y. Mu, and W. Fan. Supervised deep learning with auxiliary networks. In *Proc. of KDD*, New York, 2014.

[137] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ASPLOS*, 2015.

[138] L. Zheng, C. Zeng, L. Li, Y. Jiang, W. Xue, J. Li, C. Shen, W. Zhou, H. Li, L. Tang, T. Li, B. Duan, M. Lei, and P. Wang. Applying data mining techniques to address critical process optimization needs in advanced manufacturing. In *Proc. of ACM KDD*, 2014.

[139] M. Zhong, K. Shen, and J. Seiferas. Replication degree customization for high availability. In *Proc. of EuroSys*, Glasgow, 2008.