

國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

異質性伺服器叢集下之工作分配與排程
Job Dispatching and Scheduling under Heterogeneous
Clusters

林廷舟
Ting-Chou Lin

指導教授：劉邦鋒博士
Advisor: Pangfeng Liu, Ph.D.

中華民國 103 年 10 月
Oct, 2014

國立臺灣大學
資訊網路與多媒體研究所

碩士論文

異質性伺服器叢集下之工作分配與排程

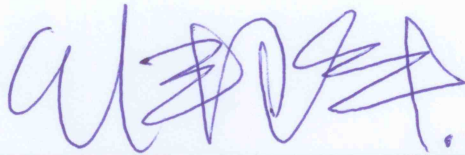
林廷舟
撰

國立臺灣大學碩士學位論文
口試委員會審定書

異質性叢集下之工作分配與排程
Job Dispatching and Scheduling under
Heterogeneous Clusters

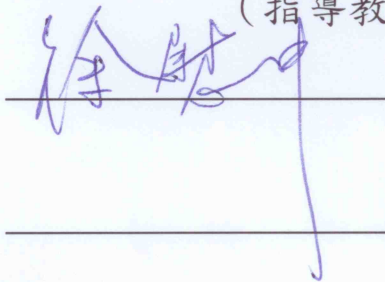
本論文係林廷舟君（學號 R01944027）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國一百零三年九月廿二日承下列考試委員審查通過及口試及格，特此證明

口試委員：



（簽名）

（指導教授）



所 長：

摘要

越來越多的企業與機構開始建造自己的資料中心作為私有雲（private cloud）使用。在這些資料中心裡，機器的性質與能力會因為新採購與被淘汰而有所不同，但由於採購通常只會在年度或半年度時進行，在這中間，機器的數量是固定的。在這樣的異質性環境下，根據不同工作（job）的資源需求與特性去進行排程，使得它們可以盡量在其要求的時間限制前完成是很重要的。

在本論文中，我們提出了一個雲端資源管理框架以動態的調配在系統中各個工作所能夠使用的運算節點（node）數量，進而達到有效率整個系統的運算資源。

關鍵字：工作排程、異質性伺服器叢集、資源分配

Abstract

Many enterprises or institutes are building private clouds by establishing their own data center. In such data centers, the physical machines can be different due to annual upgrades, but the amount of machine are fixed for most of the time. In such heterogeneous environment, scheduling jobs with different resource requirements and characteristic in order to meet different timing constraints is important.

In this paper, we proposed a cloud resource management framework to dynamically adjust the number of computation nodes for every job in the system.

keywords: Job/Task Scheduling, Heterogeneous Clusters, Resource Allocation

Contents

摘要	iii
Abstract	v
1 Introduction	1
2 Related Work	3
3 System Architecture	5
3.1 Job and Task Model	5
3.2 System Overview	6
3.3 Client	7
3.4 Worker	9
3.5 Management System	10
3.5.1 Status Checker	10
3.5.2 Decision Maker	11
3.5.3 Dispatcher	11
4 Implementation	13
4.1 System Flow	13
4.2 RPC Server Component	14
4.3 Message Service	14
4.4 Adaptive Adjustment	15

5	Scheduling Policies	17
5.1	Priority-based Policy	17
5.2	Proportion-based Policy	18
5.3	Workload-based Policy	19
5.3.1	Model Definition	19
5.3.2	Algorithm	19
5.4	Deadline-based Policy	20
5.4.1	Model Definition	21
5.4.2	Algorithm	21
5.5	Spare Resource Allocating	22
6	Experiment	23
6.1	Hardware Settings	23
6.2	Worker Failure Tolerance	23
6.3	Trace-based Simulation	24
6.3.1	Simulation Settings	25
6.3.2	Experiment Results	26
7	Conclusion	31

List of Figures

2.1	Graphical Management Tool of JPPF	4
3.1	System Architecture Overview	6
3.2	Job allocation adjustment	12
6.1	Worker Instance Allocation	24
6.2	Average Priority Violation Rate under Homogeneous Environment	27
6.3	Average Priority Violation Rate under Heterogeneous Environment	28
6.4	Deadline Miss Penalty under Homogeneous Environment	29
6.5	Deadline Miss Penalty under Heterogeneous Environment	30

List of Tables

Chapter 1

Introduction

Cloud computing has become a popular issue in recent years. Many enterprises or institutes are building private clouds by establishing their own data centers. In such data centers, the maximum amount of computing resources and the number of physical machines are fixed for most of the time. They change only during adding newly purchased servers or removal of obsolete ones during equipment upgrade.

Jobs in a data center may vary from one another. They may have different characteristic, resource requirements, time constraints, and priority. For example, scientific computation requires large computing resources, while a billing sub-system need to generate the credit-card bills for each user every month. Also, newly arrival emergency jobs may require large amount of resource in a short period. How to allocate resources for jobs in a heterogeneous environment and meet different requirements becomes an important issue.

In this paper, we proposed a cloud resource management framework to adjust the number of computation nodes for every job in the system dynamically. This framework makes decisions according to some specified policies — *priority-based*, *proportion-based*, *workload-based* and *deadline-based*. These policies take the remaining workload, priority or deadline of each job into consideration and generate a scheduling plan. Administrators can choose a different policy or specify a customized one for the system in order to fit the their needs. Moreover, this framework can work as an individual cluster management system or as extension components of an existent cloud system.

The rest of the paper is organized as followed. The system architecture is presented

in Chapter 3 and the implementation is discussed in Chapter 3. Chapter 5 introduce the four policies we implement for different types of jobs. The experimental results are in Chapter 6. Chapter 7 is the conclusion.

Chapter 2

Related Work

Job scheduling with deadlines has been a well studied field. In real-time systems, where each job has a hard deadline to be done, Earliest Deadline First (EDF) policy has been widely adapted for it is an optimal dynamic schedule policy on preemptive uniprocessors. By optimal it means if all jobs submitted to the system can be scheduled in a way that every job can be done before deadline, EDF is able to schedule them so that all of them can finish before their deadlines [?].

Various works have studied dynamic scheduling methods for parallel real-time jobs (PRJs) in clusters. Qin et al. modeled parallel real-time jobs as directed acyclic graphs and proposed a reliability-driven method [?]. He et al. represented real-time processing as hybrid execution of existing periodic jobs and newly arriving aperiodic jobs, and proposed a scheduling method by modeling spare capabilities of new arriving aperiodic parallel real-time jobs [?]. Xie et al. proposed TAPADS, Task Allocation for Parallel Applications with Deadline and Security Constraints, which takes security constraints into consideration [?, ?]. In these cases, deadlines are hard deadlines. If the system cannot guarantee a submitted job can be finished before required deadline, it is then rejected.

On the other hand, several works have focused on scheduling with soft deadlines, where missing deadlines is possible but might yield different kind of costs. Soft deadlines are also viewed as a kind of service level agreements (SLAs) or Quality of Service (QoS) demands. Service level agreements are usually applied in clusters that provide charged computation services, so lots of works took user utility as their objective. For instance,

Libra is a computational economy driven scheduling system designed to support allocation of resources based on the users' quality of service requirements [?], and Yeo et al. used a linear penalty model on user utility [?]. Also, job with QoS demands are often associated with grid computing; He et al. proposed a two-level optimization — MUSCLE, a multi-cluster level scheduler, and TITAN, a single-cluster level workload manager — for scheduling in parallel jobs in multi-clusters and grids [?]

Java Parallel Processing Framework (JPPF) [?] is a very popular open-source job dispatching framework. It has gained enormous amount of users with abundant customization APIs, user friendly management tool in GUI (figure 2.1) and being extremely easy to deploy. However, because its design is to let available nodes pull tasks from a queue, it is very hard to achieve node-aware scheduling that can assign tasks to a certain machine explicitly.

Driver / Job / Node	State	Initial task ...	Current task c...	Priority	Max nodes
lolo-i720.home:11198					
[Node id=1] (121)	Executing	4	2	0	∞
lolo-i720.home:11199			1		
[Node id=1] (125)	Executing	5	5	0	∞
[Node id=1] (129)	Executing	4	4	0	∞
[Node id=1] (133)	Executing	8	8	0	∞
[Node id=1] (137)	Executing	2	2	0	∞
[Node id=1] (141)	Executing	3	3	0	∞
[Node id=1] (145)	Executing	2	2	0	∞
[Node id=1] (149)	Executing	2	2	0	∞
[Node id=4] (120)	Executing	5	1	0	∞
lolo-i720.home:12004			1		
[Node id=4] (168)	Executing	2	2	0	∞
[Node id=4] (172)	Executing	5	5	0	∞

Figure 2.1: Graphical Management Tool of JPPF

Chapter 3

System Architecture

In this chapter, first we will introduce the job and task model in our framework. Then we give an overview of the proposed framework, followed by the description and of each component.

3.1 Job and Task Model

We defined our job and task models as the following. A *job* submitted by a user consists of several *tasks*, the minimum scheduling unit of our system. *Tasks* in the same job are independent to each other, so that can be processed in parallel. A *job* is finished if all the *tasks* from this *job* have been processed.

Here is an example of jobs and tasks. In a telecommunication company, they have to send bills to users every month. However, the settle days of users are not the same. Therefore the system will batch user data with the same settle day into a *job*, and calculate the amount of money each user has to pay. Since user data are independent to each other, they can be processed in parallel. Computation on data of a single user should be a task.

We made some assumptions about our models. We assumed that the number of tasks in a job is known, but workload of tasks in a job may differs. The remaining workload of a job can be roughly estimated by counting the number of unprocessed tasks or summing the workloads (if explicitly provided) of these tasks. A job may have a “deadline”, which means all the *tasks* of this job should be finished before this time constraint, or it may

cause penalty or costs otherwise.

3.2 System Overview

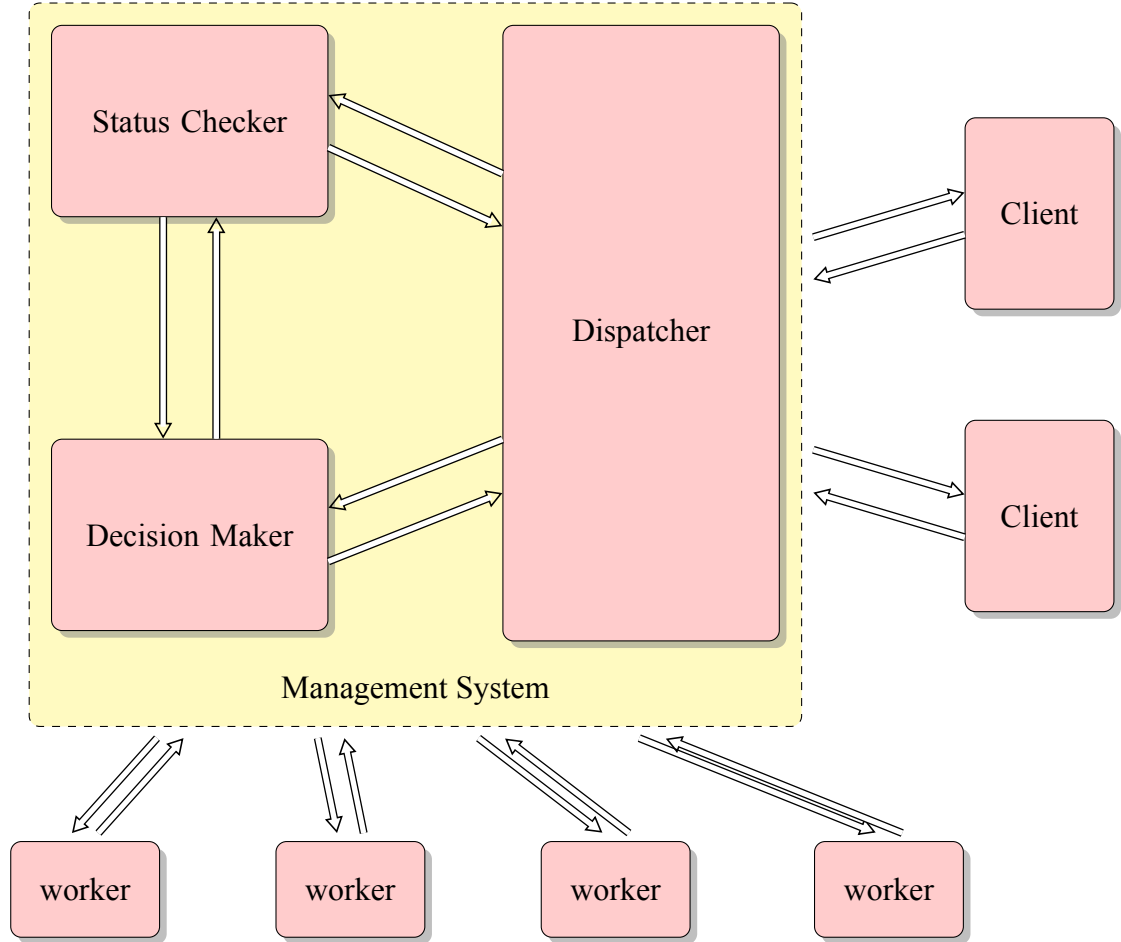


Figure 3.1: System Architecture Overview

Our cloud management framework consists of three parts: the *management system*, *clients*, and *workers*. The *management system* is the major part of our framework. Users submit jobs from *clients*. *Workers* are the computation nodes that process *tasks* in *jobs*. Figure 3.1 gives the sketch of the architecture of the proposed framework. The framework can work as an individual cloud computing system or extension components of an existent cloud system.

3.3 Client

The *client* is the programming interface for users to submit jobs to our system. In our framework, a user will submit *jobs* to the system via a client instance created using the provided library.

Users specify several attributes to jobs before submission, such as deadline, priority and execution profile from previous experience. These attributes are later passed to decision maker for reference so that it can schedule the jobs.

The client library also provides several features to end users to satisfy their varying programming needs. First, we support background task execution. Users can specify the synchronization point that waits all the submitted task to be done as they like, without blocking the code before it. Besides, user can submit multiple jobs at a time.

Combining these two features, user can easily program the synchronization model they want, just like those traditional work flow description languages can do [?]. Here's an example on how to program with our client library.

```

client = Client.new
client.register(SERVER_ADDRESS)
client.start

j1 = Job.new('Job1')
j1.add_task Task.new(...)
... # Add more tasks

j2 = Job.new('Job2')
j2.add_task Task.new(...)
... # Add more tasks

# 200-second deadline
j1.deadline = j2.deadline = Time.now + 200.0

# Submit j1 and j2 together
# Background execution
j12_waiter = client.submit_job([j1,j2])

# Do other time consuming computation
...

j3 = Job.new('Job3')
j3.add_task Task.new(...)
... # Add more tasks

# Remaining part can't run until j3 is done
j3_waiter = client.submit_job(j3)
client.wait(j3_waiter)

# Some more things to do
...

# Wait until j1 and j2 is done.
client.wait(j12_waiter)

# Combining j1, j2 and j3
...

```

Example Code 1: Sample code of client usage

Assume that the user has 3 jobs — j_1 , j_2 and j_3 — to be done separately, and their we

result will be aggregated afterward in the following scenario: j_1 and j_2 takes long time to run on the cluster can be submitted directly; j_3 takes shorter time on remote clusters if separated into parallel tasks, but requires time consuming local pre-processing and post-processing. Since j_1 , j_2 and the pre-processing of j_3 is costive, it is better to do them in parallel. As a result, we should submit j_1 and j_2 at first and wait for their results in background, and then start the pre-processing of j_3 . Not needing post-processing, results of j_1 and j_2 are synchronized after j_3 . Finally, as all of the results come back, we can do the final aggregation.

3.4 Worker

Workers are the computation nodes that consume cloud resources to process tasks. They execute tasks assigned by the management system. A worker instance executes at most one *task* at a time. In other words, a worker is the minimum scheduling slot of our system. However, it does not imply that a physical machine can run only one task at a time. A physical machine can serve multiple worker instances, therefore executing more than one task simultaneously.

Deploying multiple worker instances on a powerful machine (e.g., with large number of cores) can gain better performance from multiprogramming, but from another aspect, it might however cause resource contention on low-end machines. We leave the choice to system administrators.

If the task execution performance of a machine running multiple worker instances is far worse than expected, it is very likely due to resource contention. In that case, the system administrator should consider reducing the number of worker instances on that machine. The pluggable implementation enables the reduction to be done *online*, i.e. without stopping any of other components.

3.5 Management System

The core part is the management system, which consists of three components: *status checker*, *decision maker* and *dispatcher*. *Status checker* periodically collects the information about *worker*, and sends these information to *Decision Maker*. *Decision maker* is in charge of adjusting resource allocation plans. *Dispatcher* is the component that deal with the physical resource allocation adjustment according to the allocation plan made by the decision Maker.

3.5.1 Status Checker

Status checker periodically collects the information about worker instances and physical servers. Exploiting the information collected by status checker, the decision maker can make resource allocation plans according to a policy specified by the system administrator.

The most essential information for the system is probably the status of workers. A worker can be in status of either *available*, *occupied*, *busy* or *down*. *Available* means the worker is idle and ready to accept task assignment, *occupied* indicates that the worker is assigned to a job but executing any task (e.g., waiting for necessary data), *busy* is the state that a worker is really executing a task and *down* shows that the worker instance is currently nonfunctional.

Aside from essential worker status, system administrators can also specify other kinds information to collect if it's essential to their customized policy. For example, if someone deploys the management system on a cluster which focuses on utilizing its I/O bandwidth, they would probably specify a schedule policy in regard to I/O utilization of the physical server. In this situation, status checker must collect the information about not only the worker instances but the physical server as well.

Necessary information will be written to a database for persistent storage. This allows status checker to recover its state and work normally after being plugged off. Besides, writing to a external database makes integration with other framework easier since it's more general to access a database.

3.5.2 Decision Maker

Decision maker is in charge of adjusting resource allocation plans. It makes allocation plans according to a specified *policy* that takes different parameters, for example, job deadline and priority, into consideration.

Instead of letting decision maker schedule resources actively, we decided to have it work in *passive mode*, which means it is invoked only under certain circumstances. Normally, it is only invoked by dispatcher whenever a job is submitted or finished. By this design, decision maker is only responsible for making allocation plans. In other words, the only job this component do is to perform the schedule algorithm and give results.

Doing seemingly little work only, decision maker is however the most critical part of the management system. This is not only because the whole system acts according to the plan the decision maker makes, but also because it provides the flexibility that other frameworks can easily obtain the result of the scheduling algorithm by invoking decision maker.

3.5.3 Dispatcher

Dispatcher is the component that deal with the physical resource allocation adjustment according to the allocation plan made by decision Maker. It receives job requests from clients, sends the job information to decision Maker, and coordinates the workers and clients to run the jobs according to the policy made by decision Maker.

Figure 3.2 is a nice example of adjusting resource allocation: There were 10 workers ($w_{1\sim 10}$) managed by the system. In the beginning, two jobs J_1 and J_2 had been allocated with 5 workers respectively ($w_{0\sim 4}, w_{5\sim 9}$), as shown in the upper half of figure 3.2. After J_3 with higher priority specified than J_1 and J_2 specified, had been submitted, decision maker computed a new allocation plan according to a specified policy. In the example, the new allocation plan was 3 workers for J_1 and J_2 each ($w_{0\sim 2}, w_{5\sim 7}$) and 4 workers ($w_{3,4,8,9}$) for J_3 . Therefore, after w_3, w_4, w_8 and w_9 had finished there task on hand, dispatcher would told them to run tasks of J_3 and the allocation plan became the lower half of figure 3.2.

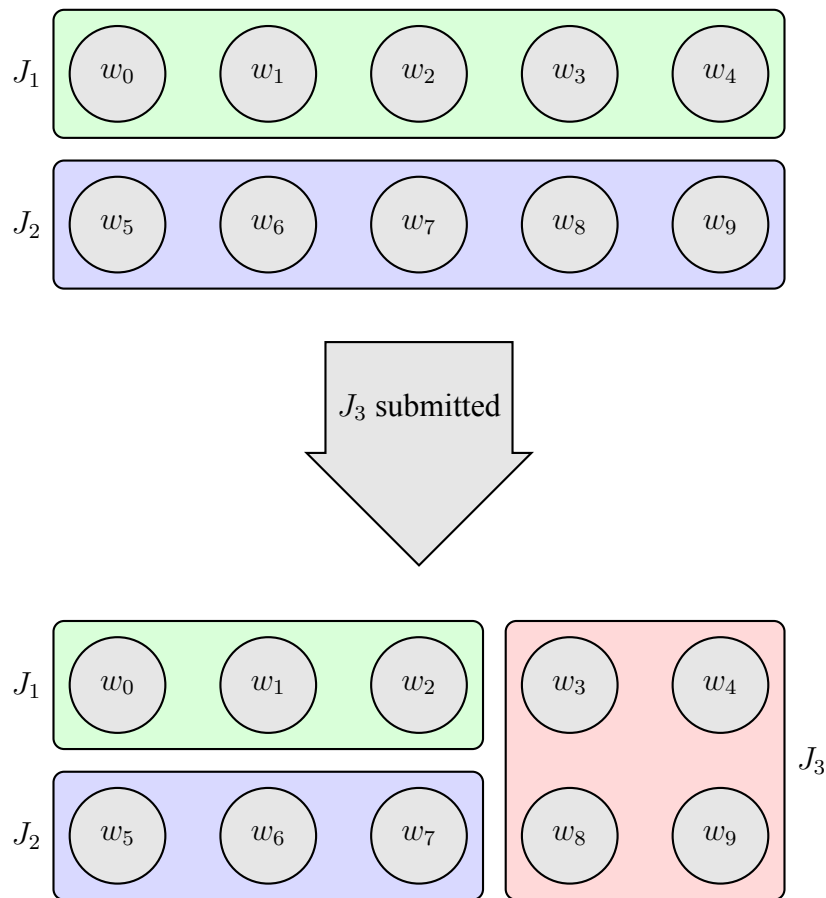


Figure 3.2: Job allocation adjustment

Chapter 4

Implementation

4.1 System Flow

Figure 3.1 shows our system flow. Each arrow on the figure indicates the communication between the components.

1. The client sends a job request to the Dispatcher in the management system.
2. According to the pre-defined policy and current worker status, decision maker generates a resource allocation plan.
3. Once the worker is ready, dispatcher responds to the client about which worker and corresponding job the client should execute next task on.
4. The client transfer the required data to workers.
5. The client sends tells the management system a task is sent ,maintaining remaining tasks of each job is critical to scheduling.
6. After finishing the task, the worker sends “job done” signal to the management system.
7. On receiving the “job done” signal, the management system informs the corresponding client.
8. The client collects the job results from workers, and

9. send an acknowledgement to workers after the transmission is completed.
10. Last, the worker send a “clear” signal to management system.

Aside from the normal job execution flow, status checker will periodically collects the system information for scheduling, including detecting worker healthiness. The scheduling process only happens whenever there is

1. job change, including submission or completion, or
2. worker change, including worker failure or rejoin.

4.2 RPC Server Component

We implemented the components in the management system and the worker as separate RPC (remote procedural call) [?] servers. Separate RPC server implementation makes each component *pluggable*. The pluggability gives the system administrator flexibility to choose the most suitable component implementation to fit different needs. Moreover, without shutting the whole system down, it is possible to change system configurations — or even upgrade the system — by substituting target components with feasible ones. Besides, this design allows us to easily integrate the management system with other cluster management frameworks like JPPF [?] or cloud operating system like Roystonea [?]. Roystonea also benefits from the RPC server implementation.

4.3 Message Service

Our implementation is a pure client-server model, which means the management system and workers cannot actively contact the clients. However, it is essential for workers and the management system to notify clients to submit tasks or get back execution results as soon as a task is done.

It is extremely inefficient for both side and server side to keep the connection between the client and the management system along the period. The connection may last over

hours if we keep it alive between task submission to worker and the finish of task execution, depending on how long a task is required to execute. Moreover, as the number of long-last connections increases, the connection number limit from the operating system will become the bottleneck of the management system.

In order to overcome this, we implemented a underlying message push service between client and the management system to allow pushing messages to clients from server side. The message service is implemented using a technique called long polling [?] — each message pull request waits on a message queue which will block the pop request on empty until timeout threshold exceeds. This implementation gives the balance between the CPU cost of frequent polling and the cost of maintaining long connections as notification channel.

4.4 Adaptive Adjustment

As we have the status checker periodically collecting system information and runtime information about tasks, we can use them to help to improve scheduling accuracy or adapt to accidental events.

For example, some scheduling policies would require to have expected runtime of a task, which is sometimes unavailable. In that case, it can use the default value first, and after several tasks are done, we can provide the logged statistics for scheduling. Moreover, machine performance degrade or temporary network congestion may lower the accuracy of expected runtime of jobs given by user. Leveraging the runtime statistics can improve the quality of scheduling.

Another case is about worker failure. In data centers, workers may fail or recover any time. Worker failure will cause execution progress delay and very possibly will result in deadline miss. In that case, if it is able to recover failed workers, the schedule plan should change and give more resource to delayed jobs (if possible) so that it may be able to meet the deadline.

To achieve this, while collecting informations, status checker also asks decision maker to schedule with updated information so that the system schedule plan can catch up with current conditions.

This feature not only improves fault tolerance, but also let us choose a more aggressive scheduling policy. Originally, the scheduling policy should in order to handle emergency jobs. This critically lowers the throughput of the system. Applying adaptive adjustment, the scheduling policy can aggressively allocate all the resources without preserving them. If emergency job comes, all we need is kill some low priority running tasks and reallocate them to the high priority emergency ones. As for the killed ones, it is just like handling worker failure; they will be redo when there are available workers allocated for them.

Chapter 5

Scheduling Policies

In this chapter, we introduce the four policies in our system — *priority-based*, *proportion-based*, *workload-based* and *deadline-based*. Each policy requires different parameters such as job priority, execution efficiency per different server and execution deadline. *Decision maker* makes dispatching plans according to a specified policy.

When designing the schedule policies, we should take the number of tasks in a job into consideration, since we don't want to assign workers more than tasks of a job to it. Recall that it is the end users' responsibility to split a job into well balanced tasks; the more balanced the tasks, the better the system schedules.

5.1 Priority-based Policy

Priority-based policy distributes workloads according to job priority. We first sort all jobs in the system according to their priority in descending order. Next, starting from the job with the highest priority, we assign x workers to the job, where x is the maximum of 1) number of idle worker and 2) number of tasks in the job. The process is repeated until there are no idle workers or all the jobs are assigned with sufficient workers. The idea behind this policy is to make the job with highest priority runs as fast as possible. However, this assignment might cause *starvation* to jobs with low priority.

To avoid *starvation*, we preserve a portion for low priority jobs. We set a reservation rate $r \in [0, 1]$ that the portion r of workers will be reserved for low priority jobs. The job

with highest priorities can use at most the $1 - r$ of all workers and the remaining workers, including those are not reserved but still not assigned to the highest priority job because of the task amount limit, will be assigned to jobs with lower priority, in the manner that one job receives one worker. The complete algorithm is shown as algorithm 1.

Algorithm 1: Priority-based policy

Input: $workerSet = \{w_1, w_2, \dots, w_m\}$, $jobSet = \{j_1, j_2, \dots, j_n\}$, preserving rate $r \in [0, 1]$

Output: A mapping of each job to scheduled workers

```

1  $result \leftarrow \text{KeyValueMap}(\text{key} \rightarrow \{\} \text{ for all key} \in jobSet);$ 
2  $jobSet \leftarrow \text{sortByPriority}(jobSet);$ 
3  $c \leftarrow \min(jobSet[0].totalTask, workerSet.size - \text{floor}((1 - r) * workerSet.size));$ 
4  $result[jobSet[0]] \leftarrow \{w_1, \dots, w_c\};$ 
5 for  $i = 1$  to  $\min(m - c, n)$  do
6   |  $result[jobSet[i]] \leftarrow \{w_{c+i}\};$ 
7 end
8 return  $result;$ 

```

5.2 Proportion-based Policy

The main concern of the proportion-based policy is the workload proportion of a job to all the others. In contrast to priority-based policy, this policy does not take job priorities into consideration unless the priority of a task is strongly related to its workload. It first sums all the workloads from each job, then compute the workload ratio of the individual job. The number of workers a job is assigned equals to the number of all workers multiplies it ratio. This algorithm, as shown in algorithm 2, is also known as *fair share scheduling* [?] that workers a job should be allocated is in proportion to its workload proportion to all jobs.

Proportion-based is a suitable solution for streaming jobs or jobs with dependencies. For streaming jobs with vary input rate, the number of workers needed is highly related to its input rate, thus workload. As for jobs with dependencies, a job may need the output from other jobs as input. Therefore, we should assign more workers to the predecessor in the beginning, and reduce the number of workers according to time. On the other hand, the number of worker of a successor will be increasing since the workload from the predecessor increased.

Algorithm 2: Proportion-based policy

Input: $workerSet = \{w_1, w_2, \dots, w_m\}$, $jobSet = \{j_1, j_2, \dots, j_n\}$,
Output: A mapping of each job to scheduled workers

```
1  $result \leftarrow \text{KeyValMap}(\text{key} \rightarrow \{\} \text{ for all } \text{key} \in jobSet);$   
2  $jobSet \leftarrow \text{sortByPriority}(jobSet);$   
3 for  $i = 1$  to  $m$  do  
4   break if  $workerSet.size$  equals to 0 ;  
5    $c \leftarrow \min(\text{ceil}(\frac{jobSet[i].workload}{\text{total workload}})), workerSet.size) ;$   
6    $result[jobSet[i]] \leftarrow \{\text{first } c \text{ element of } workerSet\};$   
7    $workerSet \leftarrow workerSet \setminus result[jobSet[i]];$   
8 end  
9 return  $result;$ 
```

5.3 Workload-based Policy

Similar to the *proportion-based* policy, the *workload-based* policy takes the workload of jobs as scheduling metric. However, workload-based takes worker heterogeneity, i.e. each worker may have different computing ability, into consideration. This policy tries to finish every job by assigning jobs to the most suitable workers, i.e. workers with higher throughput.

5.3.1 Model Definition

We represent each job j_i submitted as $j_i = (w_i, p_i, n_i)$, where w_i is the estimated workload, p_i is the job priority, and n_i is the number of tasks in j_i . As for the workers, we model each worker s_k as a vector of estimated *throughput* of each submitted job; we can write it as $s_k = (th_1^k, th_2^k, \dots)$.

5.3.2 Algorithm

We apply a greedy approach in this policy. First, we sort all the jobs according to their priority in descending order. Starts from the job with the highest priority J_m , sort the remaining workers according to their throughput th_m to J_m . Then assign the top k workers that can satisfy the workload requirement of J_m , and remove assigned workers from the list. The job is also removed from the list waiting for assignment. Repeat this

process until there are no idle worker or unassigned job. The complete algorithm is shown as algorithm 3.

Algorithm 3: Workload-based policy

Input: $workerSet = \{w_1, w_2, \dots, w_m\}$, $jobSet = \{j_1, j_2, \dots, j_n\}$,
Output: A mapping of each job to scheduled workers

```

1  $result \leftarrow \text{KeyValueMap}(\text{key} \rightarrow \{\} \text{ for all key} \in jobSet);$ 
2  $jobSet \leftarrow \text{sortByPriority}(jobSet);$ 
3 for each job  $j_i = (w_i, p_i, n_i) \in jobSet$  do
4   if  $workerSet$  is empty then
5     break;
6   end
7    $cmp \leftarrow \text{function}(w) \{ \text{return } w.th[j_i] \};$ 
8    $workerSet \leftarrow \text{sort}(workerSet, cmp, DECESENDING);$ 
9    $th \leftarrow 0;$ 
10  for ( $k = 0;$ 
11     $k < workerSet.size$  AND  $k < n_i$  AND  $th < w_i$ ;  $k \leftarrow k + 1$ ) do
12     $th += workerSet[k].th[j_i];$ 
13  end
14   $result[j_i] \leftarrow workerSet[0..i];$ 
15   $workerSet.remove(0, k)$ 
16 end
17 assign remaining workers to jobs still in need;
18 return  $result$ ;
```

5.4 Deadline-based Policy

Deadline-based policy tries to finish every jobs before their soft deadline. If a feasible scheduling plan does not exist, this policy tends to meet the deadlines for jobs with higher priority. There are varies kinds of jobs that must be finished before a given time constraint. For example, as a Internet service provider, settling monthly bills of millions of users within few days after the monthly charge-off day is very critical to their business. It is obvious that computing resource allocated to this kind of job should increase as their deadline approaches — the closer the deadline is, the more worker a job should get. Unfortunately, policies introduced above does not directly adapt to this need; hence, we introduce this policy designed for *deadline-aware* scenarios.

5.4.1 Model Definition

For this policy, we assume that each job is provided with priority and deadline. The execution time for each job is assumed to be known, which can be estimated by historical data or profiling. A job j_i can thus be represented as $j_i = (d_i, p_i, n_i)$, where d_i, p_i, n_i refers to the job deadline, job priority, and the number of tasks in j_i , respectively. The workers are modeled as a vector of estimated execution time of each submitted job: $s_k = (T_1^k, T_2^k, \dots)$, similar to the previous policy.

5.4.2 Algorithm

A job should be finished before its soft deadline. Intuitively, for the same job, if the deadline is twice tighter, it requires twice as many workers to meet that deadline. More generally speaking, the deadline and the workload are in reciprocal relationship. This implies that an worker allocation S to job j_i meeting the deadline is equivalent to

$$\sum_{s_k \in S} \frac{1}{T_{j_i}^k} \geq \frac{1}{d_i}$$

With this observation, only a little modification on the *workload-based* policy is needed to apply to this model: simply use the inverse of deadline as throughput.

First, sort all the jobs according to their priority in descending order. Starting from the job with the highest priority J_m , sort the remaining workers according to their throughput th_m to J_m . Then assign the top k workers that can finish J_m before its deadline, and remove assigned workers from the list. The job is also removed from the list waiting for assignment. Repeat this process until there are no idle worker or un-assigned job. The complete algorithm is shown as algorithm 4.

5.5 Spare Resource Allocating

Originally, scheduling policies should consider preserving resource for possible emergency jobs like workload-based or deadline-based policy. With the adaptive adjustment

Algorithm 4: Deadline-based policy

Input: $workerSet = \{w_1, w_2, \dots, w_m\}$, $jobSet = \{j_1, j_2, \dots, j_n\}$,
Output: A mapping of each job to scheduled workers

```
1  $result \leftarrow \text{KeyValMap}(\text{key} \rightarrow \{\} \text{ for all } \text{key} \in jobSet);$   
2  $jobSet \leftarrow \text{sortByPriority}(jobSet);$   
3 for each job  $j_i = (d_i, p_i, n_i) \in jobSet$  do  
4   if  $workerSet$  is empty then  
5     break;  
6   end  
7    $cmp \leftarrow \text{function}(w) \{ \text{return } 1/w.execTime[j_i] \};$   
8    $workerSet \leftarrow \text{sort}(workerSet, cmp, DECESENDING);$   
9    $th \leftarrow 0;$   
10  for ( $k = 0;$   
11     $k < workerSet.size$  AND  $k < n_i$  AND  $th < 1/w_i.execTime[j_i]; k \leftarrow k + 1$ )  
12    do  
13       $th += 1/workerSet[k].execTime[j_i];$   
14    end  
15     $result[j_i] \leftarrow workerSet[0..i];$   
16     $workerSet.remove(0, k)$   
17 end  
18 assign remaining workers to jobs still in need;  
19 return  $result;$ 
```

in section 4.4, we can aggressively allocate the spare resource as much as possible. There are 2 ways to do so, allocate them by priority or by closest deadline. For jobs still requires worker to process, sort them by the criteria (priority or deadline), and allocate the rest workers one by one until workers are used up or every job has obtained enough workers. This is the additional optimization we did on the policies that preserves resource in advance for emergency jobs like workload-based or deadline-based policy.

Chapter 6

Experiment

6.1 Hardware Settings

Our experimental environment consists of one master node and ten working nodes. The master node is a physical machine with two quad-core CPU. The CPU model is Intel(R) Xeon(R) CPU X5570 @ 2.93GHz. On the other hand, each working node is a single-core physical machine. The memory sizes are 1.5GB and 768MB, respectively. Our management system works on the master node. The master node also serves as a client which generate the workloads. There are two workers on each working node.

6.2 Worker Failure Tolerance

We reproduced a small scale workload trace provided by Chunghwa Telecom Laboratories [?]. Total of 8 worker instance is used and deadline-based scheduling policy is applied in this simulation. Only 1 job was sent and the deadline is set to 4 minutes. The characteristic of this workload is that it starts with several light-weight tasks and then comes with heavier tasks.

Figure 6.1 shows the worker allocated for that job. The blue line is the normal execution, and the red line shows the execution progress with workers manually shutted down to simulate worker failure. Six workers are manually shutted down at 0:00:55 and resumed after 0:01:55, which means at the period, only 2 workers are available.

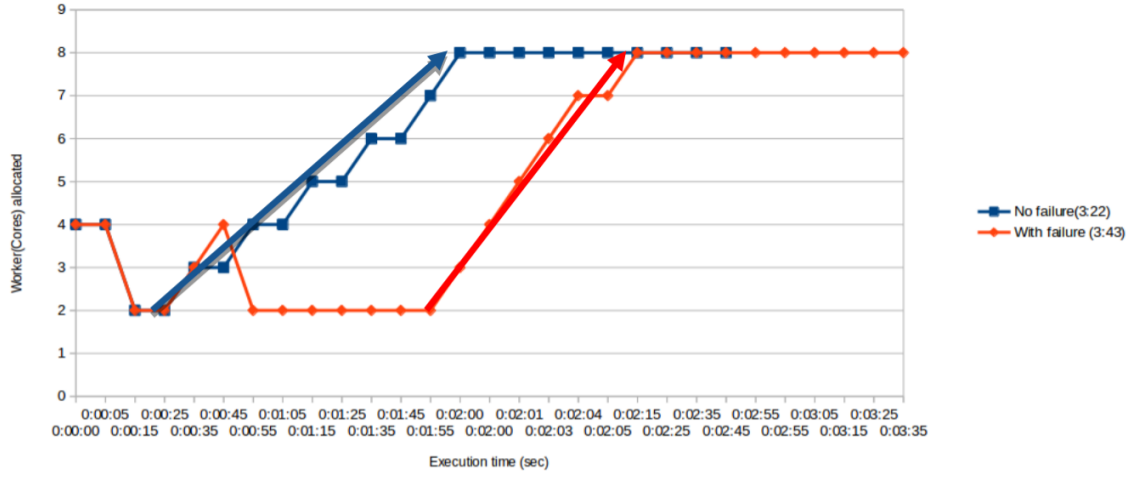


Figure 6.1: Worker Instance Allocation

We will discuss the normal execution (blue-line) first. At the beginning, since we don't have any run time information, the scheduler just use the default value (4 workers). After few seconds, some tasks are done and the run time statistics from *status checker* shows that it seems that each task execution progress is going faster than expected, so the system decided to decrease the number of scheduled workers. However, the system later found that the task execution progress had been slowing down and current allocation might not be able to meet the deadline, so it schedules more workers to the job.

The simulation with worker failure is similar to the normal one except for we manually shutted down and resumed workers. After worker resumed, the system tried harder to catch up the deadline. We can see that the slope of the worker number change of red line is greater than the blue line, implying the systems ability to adapt to failure events.

6.3 Trace-based Simulation

We conducted a trace-based sleep simulation to demonstrate the efficiency of our system. Whenever a worker receives a task, it simply sleeps a period of time according to the information on the received task. Because all the worker do is simply sleeping, deploying more workers than the number of CPUs on a single server makes extremely little impact to the performance.

Since we could not obtain traces with deadline, we decided to use traces in *standard*

workload format [?]. The trace we use is the SDSC-Par96 log, containing 32135 jobs. This log contains a years worth of accounting records for the 416-node Intel Paragon located at the San Diego Supercomputer Center (SDSC). The workload logs from the SDSC Paragon were graciously provided by Reagan Moore and Allen Downey, who also helped with background information and interpretation. We took samples from these eighteen-thousand jobs, and scaled the waiting and execution time of the sampled jobs.

6.3.1 Simulation Settings

Trace Sampling

The trace is however very sparse and skewed. The running time distribution of the trace is very long-tailed and more like a exponential distribution: Enormous number of jobs have quite short running time (several seconds), while some jobs have extremely long running time (several days). Therefore, we scaled down the running time of jobs by factor of 0.001 to lower the simulation time. Besides, if we directly sample 1 percent of jobs the waiting time between jobs can still be long — maybe up to hours. Because of this, we instead sample consecutive jobs only and scaled the waiting time between jobs. Limiting the number of jobs to sample out, we randomly pick a starting job and take jobs consecutively after it. The sample rate is 0.01 and the scale factor of waiting time between jobs is 0.1.

There are still some assumptions to be made to fit our usage model. First, we cannot obtain information about subtasks of a jobs. So we take the processors of allocated for a job as its number of tasks and take the total running time as the execution time of each task, which means we assume every task in a job takes identical time to run. Besides, the information about batches, priority and deadlines is absent in the trace but however our targeted characteristics. Since jobs in standard workload format traces are independent, we group jobs that will be submitted within 1 second together to simulate batches. For priority, we group jobs with same user IDs as same priority and we give the priority group with less jobs higher priority. As for the deadlines, we use the double of its running time as deadline for those jobs with less than 4 tasks, and for those jobs with more tasks, we

use $2 * runtime * \#task / 4$ as deadline.

During the simulation, the client starts new jobs according to the arrival time and execution time from the trace. Since the actual workloads is not available, a worker will be set to “sleep mode” after receiving a task from the client. After resumed from the sleeping mode, the worker sends a message to the management system indicating it has finished a task.

Homogeneous/Heterogeneous Environment

The simulation is conducted under 2 different environment settings — homogeneous and heterogeneous. By homogeneous we mean each worker has identical performance. Under this setting, when a worker received a task, it simply sleeps the execution time marked on the task, so if 2 workers received a task respectively from a same job, the sleeping time will be the same. In contrast, in heterogeneous environment, workers may have different performance.

6.3.2 Experiment Results

We evaluated the scheduling policies with two measures — priority violation rate and deadline miss penalty. These measures stands for two very diverse aspect of measuring performance of scheduling policies. We will introduce them later respectively. The target we compared with is the Earliest Deadline First (EDF) algorithm, an optimal scheduling policy in real-time systems under preemptive uni-processors.

Priority Violation Rate

In enterprise private clusters, jobs may comes in with different priority from different users sharing the same clusters. When the cluster resource is insufficient to satisfy all the jobs in the system, some jobs must be sacrificed and get less resource or even starve. High priority ones are expected to always gain more resource than low priority ones. In public clusters, a cluster management system may resource to low priority jobs rather than high ones due to different reasons such as fairness or resource utilization. However,

in enterprise clusters, priority is a very serious constraint to satisfy.

In order to show how well a scheduling policy meets priority constraint, we define a measure called *priority violation rate*. It is defined as

$$\frac{\text{\#violated jobs in the system}}{\text{\#jobs still in the system}}$$

and a violated job is a job j which is not scheduled with any worker while there is still at least one job which has lower priority than j but scheduled with one or more workers. Figure 6.2 and 6.3 show the average priority violation rate during each conducted simulation of different scheduling policies.

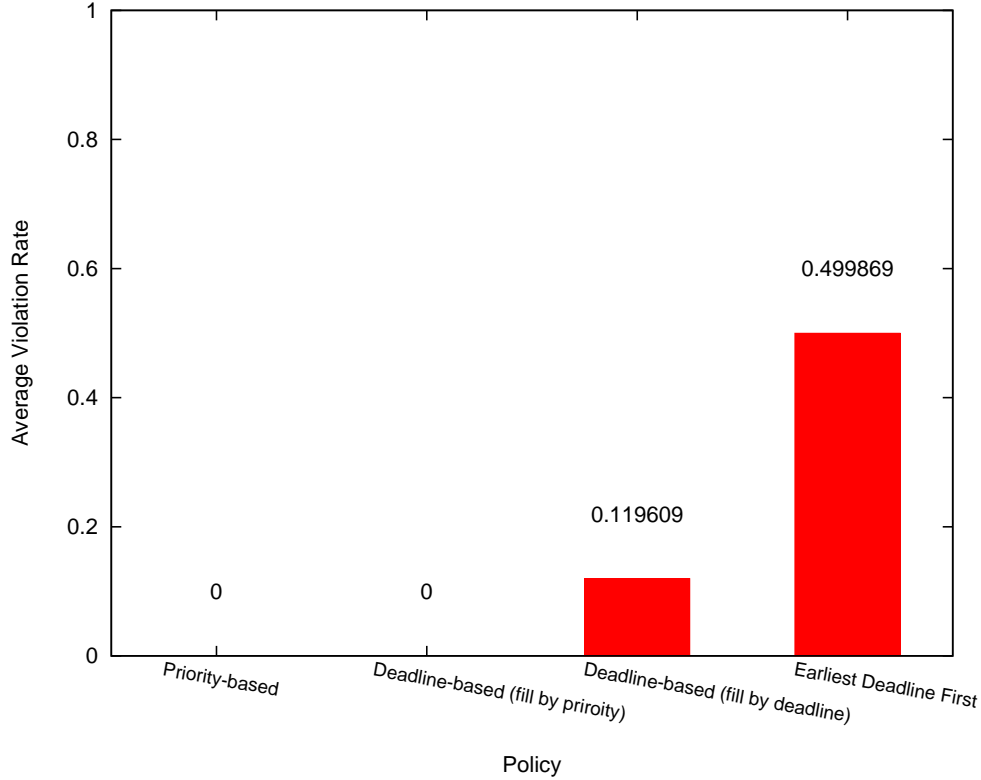


Figure 6.2: Average Priority Violation Rate under Homogeneous Environment

As we can see, under either homogeneous or heterogeneous environment setting, the average violation rate of priority based policy and deadline-based policy with spare resource filled by priority is 0, which means there is no any priority violations. High priority jobs are always scheduled before — at least together with — low priority ones under these two policies. In contrast to the previous two, deadline-based with spare resource filled by

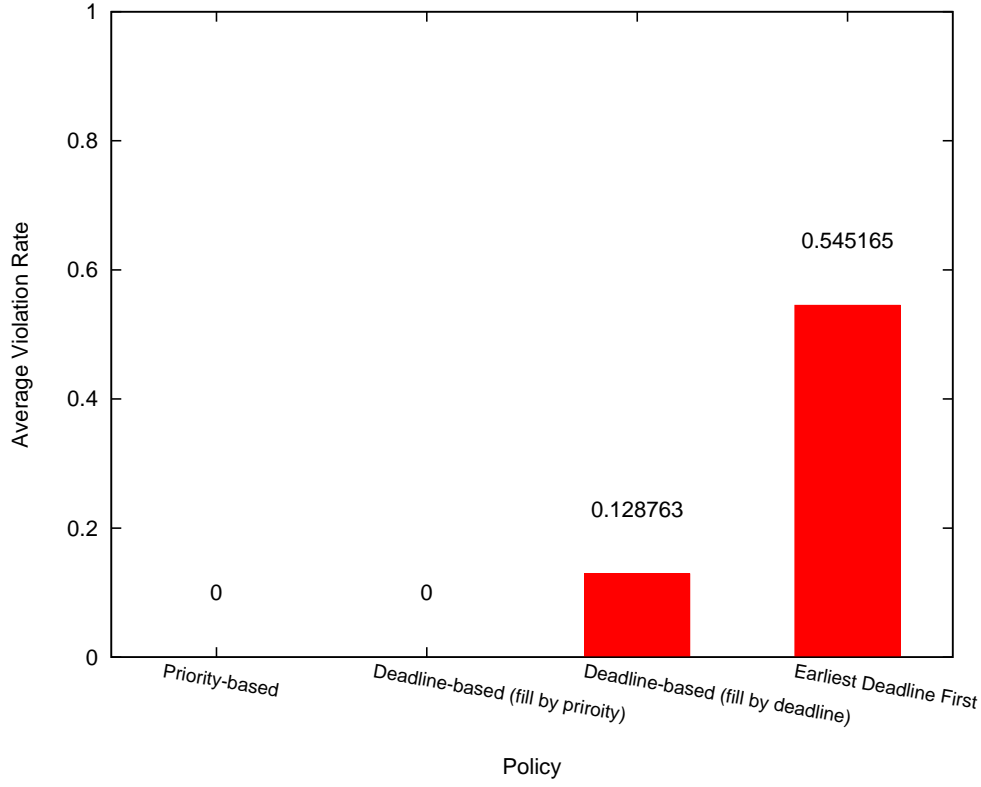


Figure 6.3: Average Priority Violation Rate under Heterogeneous Environment

deadline and EDF do not guarantee to always schedule high priority jobs before low priority ones. However, since EDF does not take priority into consideration, its average violation rate is much higher (4 times as much) than deadline-based with spare resource filled by deadline under both homogeneous and heterogeneous environment settings.

Deadline Miss Penalty

Another aspect of measuring the performance of scheduling policies in a system that takes (soft) deadlines into consideration is how many jobs did not meet their deadlines and how long did they missed. When measuring the level of the deadline missing among submitted jobs, not only the number of jobs missed deadline, but also their priority should be considered. Failing to meet deadlines of high priority jobs should be penalize than failing on low priority ones. We used $\sum_{j \in Jobs} P_j * M_j$ as our penalty measure, where j is a job of all submitted jobs, P_j is the priority of j and M_j is the deadline missed of j in seconds; we define $M_j = 0$ if a j is finished before its deadline. This measure takes both priority and deadline into consideration. The lower the penalty score, the better the

performance of the scheduling algorithm. The results are shown in figure 6.4 and 6.5. The shown result of each scheduling policy is the average of 5 sampled workloads. All of the policies used identical sample set as input benchmark.

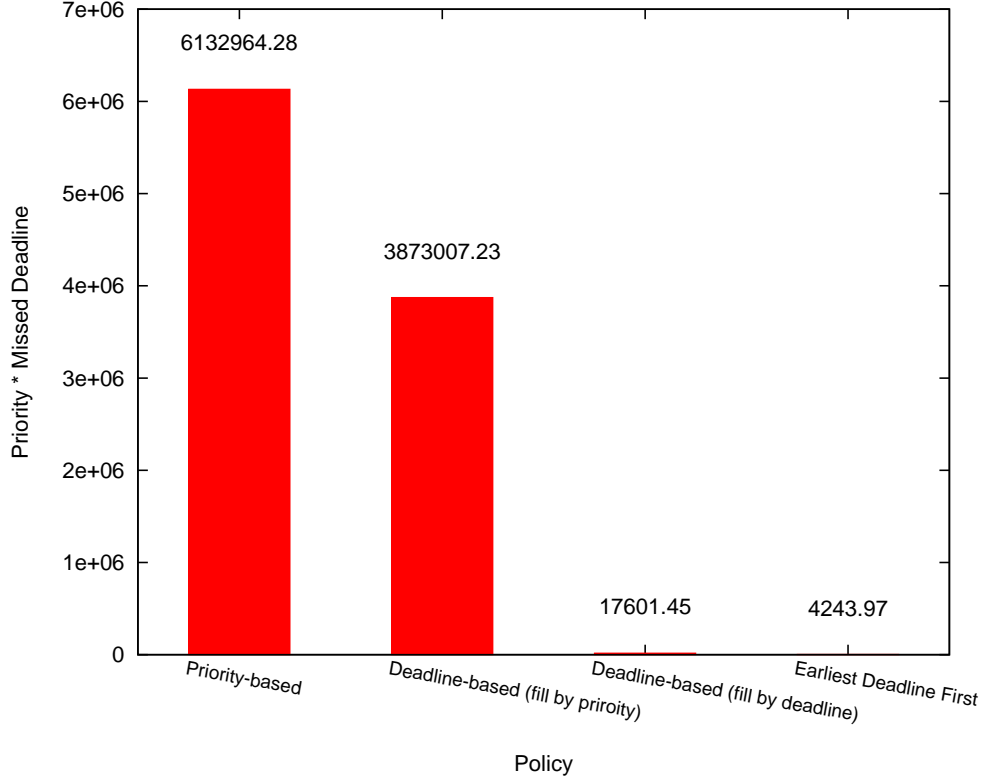


Figure 6.4: Deadline Miss Penalty under Homogeneous Environment

As we can see, EDF gives the best performance on this, either applied under homogeneous or heterogeneous environment. Priority-based policy and deadline-based policy with spare resource filled by priority did not perform well — their penalty scores are 3 magnitudes higher when compared with EDF.

Surprisingly, there is a gorgeous leap on the performance of deadline-based policy when the spare resource is assigned to jobs with the closest deadlines instead of the highest priorities. The penalty score of filling by deadline is only 0.004 of filling by priority under homogeneous environment setting, and 0.002 under heterogeneous environment setting.

We think the main reason of this is the extremely long-tailed running time distribution. Despite being scaled down, the workload still contains some very heavy-weighted jobs which takes 500 to 800 seconds. The impact made by running time (and the relative deadlines) will be much bigger compared with the priority scores limited under 100. That

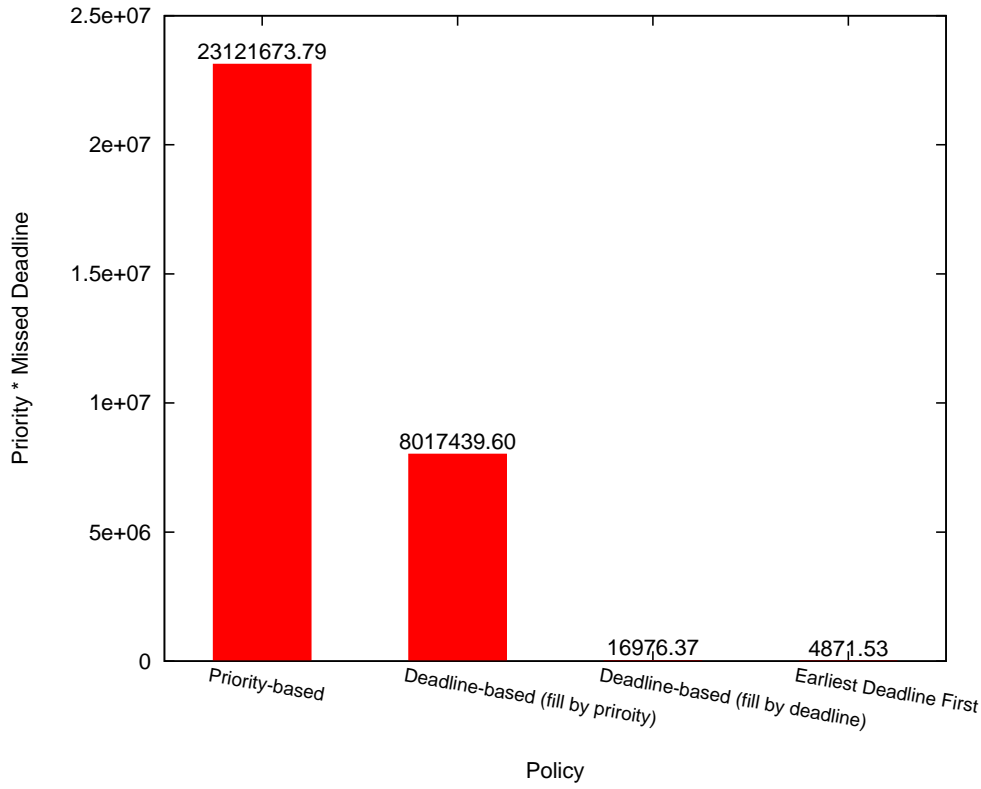


Figure 6.5: Deadline Miss Penalty under Heterogeneous Environment

is why the policies focused more on deadlines, such as EDF, gives better performance. Deadline-based policy with spare resource filled by deadline, however, attaches more importance to priority, so it slightly underperforms EDF. On the other hand, as we blended with the powerful EDF algorithm by allocating spare resource with EDF, the deadline based policy has 2 magnitudes of improvement on performance compared with filling spare resource by priority, and the performance on the deadline miss penalty is quite close to EDF.

Trade Off between the Two Measures

In figure 6.2, 6.3, 6.4 and 6.4, the policies more close to the right lay more emphasis on deadline constraints, while the ones more close to the left take priority more seriously. We found that there is a trend that the right hand side ones performs worse than left hand side ones (higher violation rate) on priority violation measure, but performs well on another measure. This somehow indicates that there is a trade off between these two measures.

Chapter 7

Conclusion

Many enterprises or institutes are building private clouds by establishing their own data center. In such data centers, the physical machines can be different due to annual upgrades, but the amount of machine are fixed for most of the time. In such heterogeneous environment, scheduling jobs with different resource requirements and characteristic in order to meet different timing constraints is important.

In this paper, we proposed a cloud resource management framework to dynamically adjust the number of computation nodes for every job in the system. This framework make decisions according to some specified policies. The proposed framework can work as an individual cloud computing system, or as extension components of an existent cloud system. Our experiment results demonstrated that our system is capable of dynamically adjusting the resource allocation plan according to the runtime statistics to meet the deadline requirements even there is worker failure. Moreover, leveraging this ability to dynamically adjust resource usage, our system can adopt more aggressive scheduling policies that do not need to preserve resources in advance and thus improves its throughput and performance.

Aside from the framework, we also proposed several scheduling policies for different situations, including 2 heterogeneity-aware ones. The deadline-based policy is a heterogeneity-aware policy that takes both priority and deadline into consideration. Our experiment shows that

1. Earliest Deadline First algorithm, an optimal solution in real-time systems, still gives the best performance on deadline miss penalty.
2. Blending the deadline-based policy with EDF by allocating the spare resource to jobs with earliest deadlines improves its performance on deadline miss penalty and almost make the performance reach the level of EDF.
3. Compared with EDF, deadline-based policy with spare resource filled by deadline outperformed a lot on meeting priority constraints with only about one-fourth of average priority violation rate.
4. There is a trade off between this priority and deadline constraints. A policy laying more emphasis on meeting priority constraints performs worse on meeting deadline constraints.

We think that deadline-based policy with spare resource filled by deadline gives a very nice balance between priority and deadline constraints.