Job Dispatching and Scheduling under Heterogeneous Clusters

Ting-Chou Lin, Pangfeng Liu
The Graduate Inst. of Networking and Multimedia
National Taiwan University
{,pangfeng}@csie.ntu.edu.tw

Ching-Chi Lin
Department of C.S.I.E
National Taiwan University
d00922019@csie.ntu.edu.tw

Chia-Chun Shih,Chao-Wen Huang
Chunghwa Telecom Laboratories
{ccshih, huangcw}@cht.com.tw

***Abstract*—...**

***Keywords*-Task scheduling;**

I. Introduction

Scalability is one of the key characteristics of cloud computing.

Virtualization wraps essential software components into virtual machines in order to make cloud systems scalable. For example, Lombardi and Di Pietro proposed ACPS (Advanced Cloud Protection System) that wraps applications, operating systems and libraries into virtual machines, and deploy them on physical servers for execution [1], [2]. After we wrap essential software components into virtual machines, we can deploy them dynamically. Consequently we can quickly adjust the number of running virtual machines to satisfy user demands.

Properly placing multiple virtual machines on a single physical machine will utilize its hardware resource more efficiently. As a hypervisor deploys more virtual machines on a physical machine, and allocates resources for them, it improves system utilization by reducing the possibility of resource sitting idle. However, as the number of virtual machines in a physical machine increases, the resource each virtual machine receives will decrease [3]. For example, if we deploy a single virtual machines on a physical machine, it can utilize all the physical resources. However, if we deploy four virtual machines on the same physical machine, each of them will receive a quarter of the hardware resource, which makes applications running on these virtual machines perform worse.

Running applications on dedicated servers can guarantee performance. A dedicated server [4] runs at most one virtual machine at a time, so that the performance of the virtual machine will not be interfered by other virtual machines.

The concept of dedicated server is feasible in data centers. The number of physical servers of a data center is usually fixed [5], and it only changes infrequently when we add newly purchased servers, or remove obsolete ones, during equipment upgrade. In other words, the number of physical servers within a data center is fixed between upgrades. Since the server configuration is fixed, we can statically map applications to dedicated servers.

Under the assumption that the number of physical servers in a data center is fixed, determining how many dedicated servers should be allocated for each job or application becomes a critical problem:

Matching application instances to dedicated servers is crucial to performance.

Jobs with different priority requiring different processing throughput will start and finish at different time; handling these varying conditions and dynamically adjusting allocated resource of each job so that every job could meet its requirement is what we want to do.

The goal of this project is to develop a cloud resource management system that dynamically adjusts the number of dedicated server a single job is granted to use according to a specified policy with regard to the remaining workload, priority and deadline of the job. The data center manager can choose a different policy to fit the their need or specify a customized policy for the system; moreover, this management system shall be able to cooperate with existing cloud operating systems.

II. System Architecture

A. Overview

We proposed a cloud resource management framework to dynamically adjust the number of physical servers for a single job. The framework make decisions according to some specified policies. The policies take the remaining workload, priority and deadline of each job into consideration, and generate a scheduling plan that satisfies the job requirements. This framework can work as an individual cloud computing system, or as extension components of an existent cloud system.

Figure 1 shows the architecture of our cloud management framework. The core part is the management system, which consists of three components: *status checker*, *decision maker* and *dispatcher*.

In the former case, the management system connects with several *workers*, which leverage physical resources directly, and interacts with instances of *client* — the interface end users submit jobs from.

We implemented the management components and the worker as separate RPC (remote procedural call) [6] servers. Separate RPC server implementation makes each component

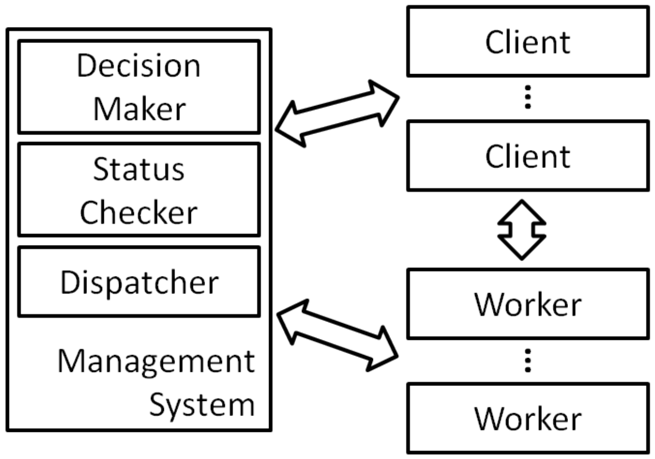

Figure 1. System Architecture Overview

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 [7] or cloud operating system like Roystonea [8]. Roystonea also benefits from the RPC server implementation.

The rest of the chapter is organized as followed. First we introduce the *client*, which is programming interface for end users to submit jobs to our system. Then we define the job model and the *worker*, the component that directly executes jobs, also the minimum scheduling slot, in our system. The three major management components — *status checker*, *decision maker* and *dispatcher* — are described after that.

B. Job and Tasks

Before discussing our system architecture, we have to introduce the representation of workload in our system. Users will submit *jobs*, each of which consists of several *tasks*, the minimum unit of scheduling, to the system. As a cluster management framework, submitted jobs are meant to be expected to be processed in parallel. It is end user's responsibility to split their jobs into balanced tasks. Just like all of other parallel computing systems, the more balanced the tasks, the better the system schedules.

C. 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, which means 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 [9]. Here's an example on how to program with our client library.

```
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
c12 = Client.new([j1,j2])
# Background execution
c12.start
# Do other time consuming computation
...
j3 = Job.new('Job3')
j3.add_task Task.new(...)
... # Add more tasks
c3 = Client.new(j3)
c3.start
# Remaining part can't run until j3 is done
c3.wait_all
# Some more things to do
...
# Wait until j1 and j2 is done.
c12.wait_all
# Combning 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 result to be aggregated: 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 pre-processing of j_3 is costive, it's better to do them in parallel. As a result, we submit j_1 and j_2 at first and wait for their results in the background then start pre-processing 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.

D. Worker

Workers are the ones using resource of the cloud directly. 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 mean that a physical machine can run one task only at a time. One can run multiple worker instances on a physical machine so that a physical machine could run 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 in contrary, 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. Thanks to the pluggable implementation, this can be done *online*, without stopping any of other components, even running workers which is preferred to keep on that machine, by simply stop several worker instances.

In fact, our management system can function without workers. In the case that it is working as a part of other cluster management frameworks, the framework in charge should be responsible for manipulating physical resources instead.

E. Status Checker

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

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; 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 deployed 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.

F. 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.

G. Dispatcher

Dispatcher is the component that deal with the physical resource allocation adjustment according to the allocation plan made by the decision maker. Figure **??** is a nice example of adjusting resource allocation: There were 10 workers ($w_{1\sim10}$) managed by the system. In the beginning, two jobs J_1 and J_2 had been allocated with 5 workers respectively ($w_{0\sim4}$, $w_{5\sim9}$), as shown in the upper half of figure **??**. After J_3, with which higher priority had been specified than J_1 and J_2, 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 ($w_{0\sim2}$, $w_{5\sim7}$) 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 **??**.

III. 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 to according to a specified policy.

When designing the schedule policies, one should take the number of tasks a job is split 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.

A. Priority-based Policy

Priority-based distributes workloads according to job priority. The idea behind this policy is to make the job with highest priority runs as fast as possible. Intuitively, we can sort the jobs according to their priority then greedily schedule as many workers as possible to each job in that order. However, this might cause *starvation* — lower priority jobs never get executed.

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: $workerSet = \{w_1, w_2, \ldots, w_m\}$,
       $jobSet = \{j_1, j_2, \ldots, j_n\}$, preserving rate
       $r \in [0, 1]$
Output: A mapping of each job to scheduled workers
1 $result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key
  $\in jobSet$);
2 $jobSet \leftarrow sortByPriority(jobSet)$;
3 $c \leftarrow min(jobSet[0]$.totalTask,
  $workerSet$.size$-floor((1 - r) * workerSet$.size);
4 $result[jobSet[0]] \leftarrow \{w_1, ..., w_c\}$;
5 for $i = 1$ to $min(m - c, n)$ do
6     $result[jobSet[i]] \leftarrow \{w_{c+i}\}$;
7 end
8 return $result$;
```

B. Proportion-based Policy

In contrast to priority-based, this policy is takes job priorities into consideration very little; instead, the main concern of this policy is the workload proportion of a job to all the others. This algorithm, as shown in algorithm 2, is also known as *fair share scheduling* [10]: workers a job should be allocated is in proportion to its workload proportion to all jobs.

C. Workload-based Policy

Similar to the proportion-based policy, the main concern of the workload-based policy is the workload of the job. The main difference is, rather than fair sharing, this policy tends to meet the workload requirement of high priority jobs.

Algorithm 2: Proportion-based policy

```
Input: $workerSet = \{w_1, w_2, \ldots, w_m\}$,
       $jobSet = \{j_1, j_2, \ldots, j_n\}$,
Output: A mapping of each job to scheduled workers
1 $result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key $\in$
  $jobSet$);
2 $jobSet \leftarrow sortByPriority(jobSet)$;
3 for $i = 1$ to $m$ do
4     break if $workerSet$.size equals to 0 ;
5     $c \leftarrow min(ceil(\frac{jobSet[i].\text{workload}}{\text{total workload}})), workerSet$.size) ;
6     $result[jobSet[i]] \leftarrow$
      $\{$first $c$ element of $workerSet\}$;
7     $workerSet \setminus result[jobSet[i]]$;
8 end
9 return $result$;
```

1) Model Definition: We represent each job j_i submitted as $j_i = (w_i, p_i, n_i)$, where w_i, p_i, n_i represents the estimated workload, priority and the number of task of j_i respectively. As for the workers, we model each worker s_k as a vector of estimated *throughput* of each submitted job; we can write is as $s_k = (th_1^k, th_2^k, \ldots)$.

2) Algorithm: The approach is a simple greedy method: First, sort the jobs by priority. For each job J_m, sort remaining workers according to its th_m, assign top k workers that just satisfy the workload requirement of J_m and remove assigned workers from the list. If workers are used up, break the loop. The complete algorithm is shown as algorithm 3.

D. Deadline-based Policy

Plenty kinds of jobs must be finished before a given deadline. 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 it obvious that computing resource allocated 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 in previous sections can't directly adapt to this need; hence, we introduce another policy designed for *deadline-aware* scenarios.

1) Model Definition: For this policy, we assume that each job is provided with priority and deadline and we have required execution time (which may be a estimated one) of a job to run on each single server. 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 deadline, priority and the number of tasks of j_i respectively, and the workers are modeled as a vector of estimated execution time of each submitted job: $s_k = (T_1^k, T_2^k, \ldots)$, similar to the previous policy.

2) Algorithm: Intuitively, for the same job, if the deadline is twice tighter, it requires twice as many workers to meet

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  $result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key $\in jobSet$);
2  $jobSet \leftarrow sortByPriority(jobSet)$;
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      $cmp \leftarrow function(w)\{$return $w.th[j_i]\}$;
8      $workerSet \leftarrow$
       $sort(workerSet, cmp, DECSENDING)$;
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 return $result$;
```

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  $result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key $\in jobSet$);
2  $jobSet \leftarrow sortByPriority(jobSet)$;
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      $cmp \leftarrow function(w)\{$return $1/w.execTime[j_i]\}$;
8      $workerSet \leftarrow$
       $sort(workerSet, cmp, DECSENDING)$;
9      $th \leftarrow 0$;
10     for ( $k = 0$ ;
11     $k < workerSet.size$ AND $k < n_i$ AND
       $th < 1/w_i.execTime[j_i]$; $k \leftarrow k + 1$) do
12         $th$ += $1/workerSet[k].execTime[j_i]$;
13     end
14     $result[j_i] \leftarrow workerSet[0...i]$;
15     $workerSet$.remove(0,$k$)
16 end
17 return $result$;
```

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

$$\sum_{s_k \in S} \frac{1}{T^k_{j_i}} \geq \frac{1}{d_i}$$

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.

IV. Experiment

A. Experimental 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.

We conduct a trace-based simulation to demonstrate the efficiency of our system. The trace we use is the *CERIT-SC workload log*, which is provided by the CERIT-SC and the Czech National Grid Infrastructure MetaCentrum. The data sets, which contains 17,900 jobs, are generated from TORQUE traces during the first 3 months of the year 2013. We take samples from these eighteen-thousand jobs, and scale the waiting and execution time of the sampled jobs. The sample rate is XXX.

During 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. The sleeping duration is equal to the task execution length. After resumed from the sleeping mode, the worker sends a message to the management system indicating it has finished a task.

B. Experimental Results

V. Related Work

VI. Conclusion

References

[1] F. Lombardi and R. Di Pietro, "Secure virtualization for cloud computing," *J. Netw. Comput. Appl.*, vol. 34, no. 4, pp. 1113–1122, Jul. 2011.

[2] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, April 2010, pp. 27–33.

[3] B. Urgaonkar, P. Shenoy, and T. Roscoe, “Resource overbooking and application profiling in shared hosting platforms,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 239–254, Dec. 2002.

[4] “Dedicated hosting service,” http://en.wikipedia.org/wiki/Dedicated_hosting_service.

[5] B. Woolley, “A framework for developing and evaluating data center maintenance programs,” http://www.apcmedia.com/salestools/VAVR-8S5KPY/VAVR-8S5KPY_R0_EN.pdf.

[6] B. J. Nelson, “Remote procedure call,” Ph.D. dissertation, Pittsburgh, PA, USA, 1981, aAI8204168.

[7] “Java parallel processing framework,” http://www.jppf.org/.

[8] C.-E. Yen, J.-S. Yang, P. Liu, and J.-J. Wu, “Roystonea: A cloud computing system with pluggable component architecture,” in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec 2011, pp. 80–87.

[9] “Workflow management coalition,” http://en.wikipedia.org/wiki/Workflow_Management_Coalition.

[10] G. Henry, “The unix system: The fair share scheduler,” *AT T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1845–1857, Oct 1984.