International Journal of Advances in Computer Science & Its Applications
Volume 6 : Issue 2 *[ISSN 2250-3765]*
Publication Date : 31 August, 2016

Virtual Machine Deployment in Cloud Computing Platform

Chun-Hung Richard Lin, Tzu-Hua Huang, and Kuan-Hua Tseng

***Abstract*—In the paper, we use Bin-Packing algorithm to solve the resource-deployment problems. We have not only discussed different Bin-Packing algorithms, but also provided the refinement method to equip with Bulk Arrivals functionality while at the same time achieving an approximate efficiency of Online Bin-Packing and the similar result of Offline Bin-Packing. Finally, trying to testify each algorithm's impact on cloud service system's performance, we used the queueing system to compare four different algorithms' impact in our model and analyze each reference data and correlations to provide a scheduling system which aligned with the need of Cluster Deployment from a developer's viewpoint.**

***Keywords*—virtual machines, docker, swarm, cloud computing, bin-packing**

I. Introduction

Nowadays the cloud computing is an essential network application, and the most attracting part of it is the virtualization of the hardware resources and the dynamic connection of Internet. Therefore, there is no need to maintain the infrastructure for users or even service provider itself. Our study discusses a more efficient way of resource management from the viewpoint of the service provider. We discovered a virtual machine resource deployment algorithm that considers the maximum number of services under fixed cost and the virtual hardware architecture of cloud computing. The traditional way to allocate the hardware resource is to quantify the hardware specifications and calculate a suitable set of hardware usage according to the requirement. On the Docker platform which we have adopted, the service, most time is one or two programs running under background, will be packed and run in a Linux Container. These services have their requirements. Therefore, we take the old bin packing problem as an example to research this deployment problem of virtual machines and discuss the management of cloud application resources nowadays.

II. Background

A. *Linux Container (LXC)*

LXC is a virtualized Linux user interface. The main goal of LXC is to create several standalone Linux systems, above a single Linux environment. By creating Container, users can get a Linux environment, which is almost the same an autonomous one, without dividing the kernel of the host operating system.

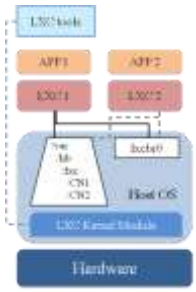

Fig. 1: LXC in Linux Host

LXC utilizes Control groups *(Cgroups)* technology and namespace to obtain independent environments with full resource accessibility at the same time. Cgroups not only integrates shared resource of the host such as CPU, memory and NIC, but also provides complete control of it. By limiting, priority, monitoring and resource isolation of each process, the Cgroup can manage resources effectively. Namespaces, on the other hand, create an isolated environment which includes the thread tree, network, user id and mounted file system so that the application can run on it.

B. *Docker and Swarm*

Docker is an open source project written by GO language as a Container management engine based on LXC. The hardware portability and platform portability can be achieved at the same time by introducing Docker engine. That means Containers can be installed on any Linux platform without additional compiler or system configuration. This characteristic makes the application which is developed based on the Docker Container, able to retrieve a running environment that is freer.

Compare to the native LXC; Docker is more isolated and lighter. Docker also supports the multilayer image file as a pattern to create the Container. When Container has built, the top layer remains writable for providing service. The Docker Container is portable and able to run on different machines.

Fig. 2: Structure of Docker

Chun-Hung Richard Lin, Tzu-Hua Huang, and Kuan-Hua Tseng
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan, R.O.C.

SEEK DIGITAL LIBRARY

Docker Swarm is one open source subproject of the Docker project. The main purpose of Docker Swarm is to build a native cluster management tool and make every node of Docker host combined with a single virtual machine. Swarm is written by GO language as well as the Docker, and there is a developing version of Swarm right now. So it is not recommended for production using, but still the architecture and some characteristics are worth for further research with the growing functions and technologies in this project.

Swarm is based on the standard Docker API thus, all Docker daemon tools such as Dokku, Compose, Krane, Flynn, Deis and DockerUI, can make their connection to Docker daemon through Docker clients. With Swarm, Docker is no longer a single host functionality but controlling and deploying Containers across multiple hosts, just like a full-resourced mainframe. The modularization of Swarm can switch the scheduler model or another module to a different one like Mesos, and that also makes the underlying Docker engine not affected by Swarm so the demands can customize it.

III. Resource Allocation Algorithm

We propose a scheduler designing for allocating resource on cloud service platform. First, the resource allocation problem can be mapped to conventional bin packing problem as solution referencing, and combining DDMS, Docker platform and Docker Swarm, plus the optimization of the real-time, we conclude the flowchart of a PaaS scheduler as below.

Fig. 3: Design of cloud service scheduler

The PaaS scheduler collect all the information and arguments from components, then select the appropriate nodes to deploy Containers. Each component has its function described as below:

Priority Policy: Calculates the priority of all service requests according to their specific arguments. In our system, we evaluate the weight of a request not only by the arguments given by its initiating parameters but also by the service type, normal type and real-time type.

Object Function: When PaaS provider constructs their cloud cluster, there will be many object orientations need to be concerned, such as loading balance, fault tolerance and the maximum number of running services. Each object have some algorithms for different demands,;our system can specify the algorithm according to setting of PaaS administration.

Scheduler Strategy: The Scheduler Strategies, including the one we proposed, are imported by Docker Swarm so that the Scheduler can select an algorithm among them for best using. It is easy for registering strategy to Swarm by its modularized design.

Strategy Optimizer: In our system, we need several strategy optimizer to limit the search for ultimate, and also people can develop their optimizers for algorithms. That is why we design the optimizer as a standalone component.

A. Scheduler Design

We propose an improve schedule algorithm based on bin packing problem and cloud service model, to increase the maximal volume of service process with limited nodes. The distributed computing system Hadoop uses the Delay Scheduling algorithm to raise the data locality by pending scheduling. By choosing the task which can use the local data to deploy first, we can reduce the unfairness of resource. We take Delay Scheduling algorithm as a reference and propose our algorithm which divides into two parts, Batch scheduling and Delay Scheduling.

B. Batch Scheduling

The purpose of batch scheduling is to accumulate the service requests. We can see that there is a significant difference between numbers of the packed box, with preprocessing of sorting and without.

The benefit of online schedule is real-time calculating, once the service request has arrived, it immediately get deployed. It is reasonable to take online schedule in a small cloud computing system. However, there may be bulk amount of service requests arrived at the same time in relatively larger cloud computing system. Online scheduling is not suitable for this situation; neither is Offline scheduling. As the consideration above, we design a schedule model which is between online schedule and offline schedule. This new scheduler can produce a better result than online schedule in an acceptable delay time.

Batch Scheduling

```
1  PaaS_Scheduler() {
2      while(!ExitSignal){
3          requeue[] = get_requests()
4          if time % scheduling cycle == 0
               // strategy[0]=Best-Fit Decreasing, strategy[1]=Bin Completion.
5              schedule_algorithm(requeue[],strategies[x])
6          end if
7      }
8  }
9  schedule_algorithm(requeue[], strategy) {
10     for each request ∈ requeue, req = 1, 2, . . . , n do
11         req.order = queue_index(requeue,req)
12         req.weight = req.cpushared + req.limitedmem
13         reqspool[] = sort_assignments(reqspool[],req)
14     end for
15     nodes[], containers[] = choose_node(strategy,reqspool[])
16     if nodes[] != null
17         Deployed containers[] to nodes[]
18     end if
19 }
```

In this proposed schedule algorithm, we implementation two offline scheduling, Best Fit Decreasing for an approximate solution and Bin Complete for the best solution. For the batch purpose, all service requests will be stored in a FIFO queue according to their arrival time; we also set a

SEEK DIGITAL LIBRARY

scheduled cycle to make the scheduler wait for a fixed period then start to calculate with this accumulative requests. After the scheduler has activated, we sort all the accumulative requests in descending order then send this sorted queue to a scheduling algorithm. The detail in pseudocode is shown as above.

C. *Delay Scheduling*

The algorithm for the bin packing problem such as the Best Fit Decreasing and Bin Completion, put the larger items into the box first, then the remaining small items have to wait to be packed till there is enough space of already opened the box. This may lead to a potential problem that the small items will be late packed or never be packed into the box due to there is not enough space after the larger one has packed. At this situation, the calculation time will greatly increase. For the purpose to reach the speed of online scheduling, we took the central idea of Delay Scheduling and designed the tolerant interval to limit the calculation time to an acceptable one.

First, we sort the requests in the batch queue in decreasing, then the scheduler handles the request which has the order smaller than the tolerant interval only. For those requests over the tolerant interval, will be a move to the next batch. This Delay Scheduling can solve the starvation problem of requests with the small demand of resource. However, the tolerant interval decreases the accumulating effect of the Batch Scheduling. The more tolerant interval increases, the better result calculated, but this will lead to longer calculating time and make the system delay responded. There must be a tradeoff between these two kinds of designs.

Batch Scheduling & Delay Scheduling

```
1: PaaS_Scheduler() {
2: while(!ExitSignal){
3:         requeue[] = get_requests()
4:         if time % scheduling cycle == 0
               // strategy[0]=Best-Fit Decreasing, strategy[1]=Bin Completion.
5:             schedule_algorithm(requeue[],strategies[x])
6:         end if
7:     }
8: }
9: schedule_algorithm(requeue[], strategy) {
10:    for each request ∈ requeue, req = 1, 2, . . . , n do
11:        req.order = req.arrivaltime / scheduling cycle
12:        req.weight = req.cpushared + req.limitedmem
13:        if req.order ≤ 1st_unschedule_req(requeue).order + tolerant interval
14:            reqspool[] = sort_assignments(reqspool[],req)
15:        end if
16:    end for
17:    nodes[], containers[] = choose_node(strategy,reqspool[])
18:    if nodes[] != null
19:        Deployed containers[] to nodes[]
20:    end if
21: }
```

D. *Real-time Request*

There are two types of cloud service request, normal and real-time. The real-time request always comes with a perfect system response time. For that, our system gives this kind of request higher priority to make it response in time. Also, a special Container called real-time Container is proposed in our system to help the real-time request.

Real-time Scheduling:

We process real-time request first by adjusting the priority of it. This will also influence the calculating result of the algorithm and reduce the total capacity of the system, but the real-time restriction can be satisfied.

Real-Time First

```
1: PaaS_Scheduler() {
2:     while(!ExitSignal){
3:         requeue[] = get_requests()
4:         if time % scheduling cycle == 0
5:             realtime_req[] = get_realtime_requests()
6:             if realtime_req[] != null
7:                 schedule_algorithm(realtime_req[],strategies[x])
8:             end if
9:             schedule_algorithm(requeue[],strategies[x])
10:        end if
11:    }
12: }
13: schedule_algorithm(requeue[], strategy) { ... }
14:
```

Real-time Container:

Based on the DDMS, our system has the capability to send control message to the cluster node; this feature can also be used for setting real-time service node.

UNIX kernel gives different setting which is about the scheduling policy and process priority to real-time process. We can give the real-time setting to a real-time node which is deployed of a real-time Container and real-time request, to obtain more resource to meet the real-time restriction. This can be done by these two UNIX commands:

 # sudo docker inspect –f `{{.State.Pid}}' $CONTAINER_ID

First, we can get the PID of this Container by this command.

 # sudo chrt –r –p $priority(32~63) $PID

Then gives the real-time setting with Round Robin schedule strategy and priority to this Container. The Round Robin here is for simultaneously running of multiple Containers instead of obtaining all resource by a single Container. Moreover, the priority can be 0 to 19 for a normal user, 20 to 31 for the system, and 32 to 63 for real-time service which we are trying to adopt in this real-time system..

IV. Experiments

For quantization modulation, we represent the queueing system on a cloud computing service platform by a queue model. We can calculate the performance of this queueing system by adjusting the parameters of it.

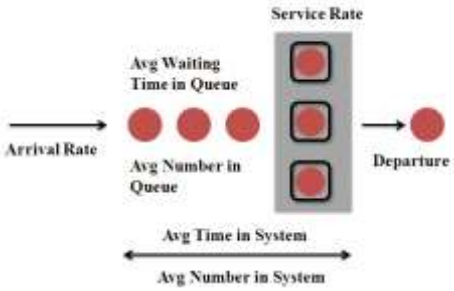

Fig. 4: Queueing System

The request with given needed resource (CPU, Memory) arrivals in this system and is put into the queue, and the scheduling system calculates and deploys it to the appropriate node which satisfies the initial arguments of this

SEEK DIGITAL LIBRARY

request and the cluster policy. After the Container has ended, the system will recover its resource and send a message to the scheduler, to form a life circle of a request. Due to the difficulty for quantization of real-time request, these experiments is only for the normal request, the real-time request is excluded.

We can user several characteristics to represent the queue model, Input Source, System Capacity, Behavior, Service Discipline and Service Facility. Then we quantify these characteristics to the arguments of the program which represented by the Kendall notation: ***A/B/C/X/Y***. ***A*** stands for the distribution of arrival time of requests; ***B*** is the distribution of service time, ***C*** is the number of the scheduler, ***X*** is the system capacity and ***Y*** is the queueing discipline.

A. Experiment Parameters

These experiments are only for the usual requests, so upon applying the Kendall notation to this queueing model, it can be represented as following: Request Arrival Rate / Service Rate / Cluster Node Number / Queueing Buffer / Schedulable. In these experiments we treat Queueing Buffer as unlimited, this makes the expression of our model as ***A/B/C/∞/Schedulable***. The parameters to this expression are ***A***, ***B***, ***C***.

In addition, to evaluate the performance of the scheduling algorithms and the scheduling designs which we have proposed at chapter 4, there are two extra parameters need to be added in, schedule cycle and tolerant interval., Therefore the total parameters to our experiments are ***A/B/C/α/β***, where

- ***A*** as average arrival time of request.
- ***B*** as average service time of a request.
- ***C*** as the number of Cluster Nodes.
- ***α*** as the schedule cycle
- ***β*** as the Tolerant Interval

B. Experiments Detail

This experiment is to evaluate the total system time and the average service time of single request on our designed scheduler. We fix the numbers of the service nodes and the number of incoming requests. Then analysis the result about the performances under the different design of scheduling algorithms.

This experiment has implemented four different scheduling algorithms and compared the differences in performance. These four algorithms are:

- Online Best Fit (Online Bin Packing):

The default algorithm of Swarm Strategy Component. In this implementation, all requests in the waiting queue will be calculated every cycle, and deploy to service node according to their arrival time. This algorithm calculates the minimal number of nodes to serve all the requests.

- Online Spread:

Online Spread is an algorithm in the Swarm Strategy Component. It is similar with the Online Best Fit except it calculates for averagely deploying the requests to nodes.

- Batch Best Fit (Batch Bin Packing):

This kernel of this algorithm is Best Fit. It has our proposed design including the concepts of batch scheduling and Delay Scheduling. The requests will be processed only when the given clock has reached. The calculation will be influenced by schedule cycle and tolerant interval.

- Batch Bin Complete:

Another implementation of our proposed design, the kernel algorithm is Bin Completion. This algorithm is for the best solution.

C. Experiment Configuration and Results

The experiment is configured as below:

- The average arrival time of request (***A***) = 2 clocks, exponential distribution.
- Service time (***B***) = 60 clocks, constant.
- Number of cluster nodes (***C***) = 5 nodes.
- Schedule cycle (***α***) = X, variable.
- Tolerant interval (***β***) = 0 cycle.
- Expression of experiment 1 queueing model: ***M/D/s/∞/Schedulable.***

The experiment results are shown below:

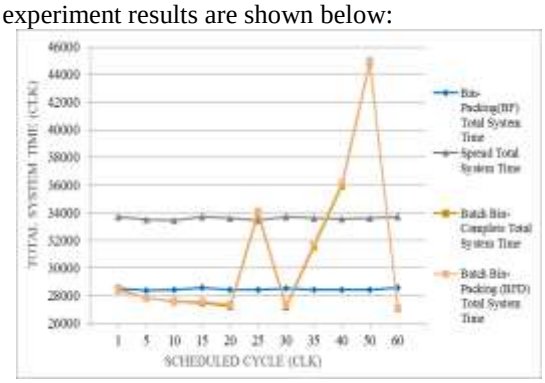

Fig. 5: Total System Time of the experiment

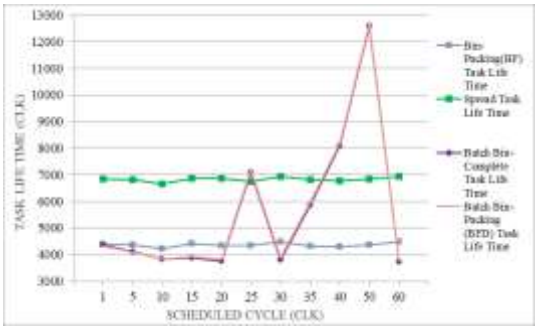

Fig. 6: Task Life Time of the experiment

At Fig. 5 and Fig. 6, we can see the two algorithms which are built-in the Swarm Strategy Component, Bin Packing and Spread, are not influenced by the schedule cycle. Moreover, these two algorithms are very differing from either total system time or task lifetime. This is due to

SEEK DIGITAL LIBRARY

the differences of purpose, online bin packing calculates the minimal number of service nodes. Instead online spread is ideal for deploy service averagely. As the result, the online spread is outperformed by the online bin packing.

Another two algorithms which contain our proposed scheduler design. We can treat the average incoming request which can be calculated by the schedule cycle (X-axis, α) / average arrival time of the request (A) as the scale of bin packing problem due to the Delay Scheduling design.

- N (Problem size) = α/A

Through observing, we can find these two algorithms can get better performance on certain schedule cycle (X-axis). Furthermore, we can get another factor which influences the result: the relationship between the service time (B) and schedule cycle (α).

Because the node resources may be released and recovered at different timing, B and α influences the utilize rate of the resource. If a resource has released before next round of scheduling, this resource will not be occupied again till next scheduling has executed. In an online system, there is no such concern because the scheduling will be executed every cycle. Instead the batch system needs to face the resource utilize rate problem. For that, we define the utility rate as below:

- $\text{Utility} = \frac{B}{\alpha \cdot \left\lceil \frac{B}{\alpha} \right\rceil}$

And calculate the utilizing rate by fixing the value of B to 60.

Resource Utility								

Table 1: Utilization rate

As we can see from Table 5-3, if we cannot make utilize rate reach to 100%, the total system time and the task lifetime will greatly increase. This research tells us that we have to pay attention to the relationship between B and α when to design a scheduler. If B is constant, then α should be a divisor of it, to make the utilize rate 100%.

- $B = c * \alpha \quad , c \in Z^+$

The next experiment is that makes B an exponential distribution, then calculate the utility rate under conditions: the total number of requests = 10000, and utility (Y-axis) value is the average value of 100 times of simulation.

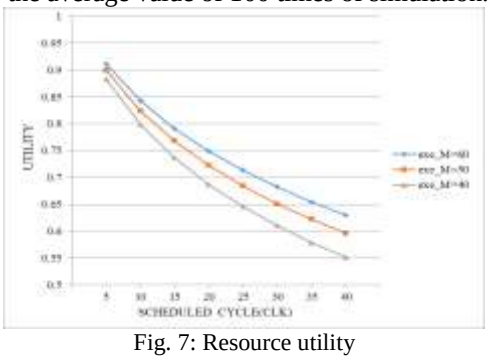

Fig. 7: Resource utility

This experiment calculates the utility under the condition that the service time (B) is given by exponential distribution. We can find that if the service time (B) is not fixed, the utility and the schedule cycle (α) are in inverse proportion. Moreover, the less the average value of service time (B), the less the utility. This is due to larger service time (B) makes the system full loading longer, as the utility is increased.

Conclusions

We take Docker engine as our foundation to implement a cloud service system and then to study the scheduling system of it. Additionally, the Docker Swarm is taken as the cluster management system. We proposed a delayed scheduling design for scheduling algorithm which modifies two kernel functions to reach the maximum service count as the target of cluster deployment. The request for real-time sustems is also considered. Then we implemented our scheduler to a cloud service system and made an experimental result of it. Finally, we compared our design with other scheduling algorithms and found the direction for improving.

On the other hand, we implement two kernel algorithms for bin packing as our delayed scheduling design in this paper; they are Offline Best Fit Decreasing and Offline Bin Completion. The Bin Completion have higher cost and higher loading due to the goal of optimization; thus, we may propose and implement a new improved, optimized design which is referred to it in the future.

References

[1] Docker -Build, Ship, and Run Any App, Anywhere, 6/2014, http://www.docker.com/.

[2] A. Bestavros, T. Cheatham, Jr., and D. Stefanescu. Parallel Bin packing using first fit and k-delayed best-fit heuristics. In Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on, 1990, pp. 501- 504.

[3] Richard E. Korf. A New Algorithm for Optimal Bin Packing. In Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, pages 731–736. AAAI/IAAI, 2002.

[4] O.R. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed priority mixed-criticality task sets. In Proc. of the 8th IEEE International Conference on Embedded Software and Systems (ICESS), 2011.

[5] Rajdeep Dua, A Reddy Raja, Dharmesh Kakadia. Virtualization vs Containerization to support PaaS. IEEE International Conference on Cloud Engineering, 2014.

[6] Jeyarani, R., Ram, R. Vasanth, Nagaveni, N. Design and Implementation of an Efficient Two-Level Scheduler for Cloud Computing Environment. IEEE, 2010

[7] Qi Cao, Zhi-Bo Wei, Wen-Mao Gong. An Optimized Algorithm for Task Scheduling Based on Activity Based Costing in Cloud Computing. IEEE, 2009

[8] Selvarani, S., Sadhasivam, G.S. Improved costbased algorithm for task scheduling in cloud computing. IEEE, 2011

[9] Mehdi, N.A., Mamat, A.; Amer, A., Abdul-Mehdi, Z.T. Minimum Completion Time for Power-Aware Scheduling in Cloud Computing. IEEE, 2012

[10] Luna Mingyi Zhang, Keqin Li, Yan-Qing Zhang. Green Task Scheduling Algorithms with Speeds Optimization on Heterogeneous Cloud Servers. IEEE, 2011

[11] Celaya, J., Arronategui, U. A Highly Scalable Decentralized Scheduler of Tasks with Deadlines. IEEE, 2011

[12] Laiping Zhao, Yizhi Ren, Sakurai, K. A Resource Minimizing Scheduling Algorithm with Ensuring the Deadline and Reliability in Heterogeneous Systems. IEEE, 2011

SEEK DIGITAL LIBRARY