Utility-based VM Assignment in Cloud Scenarios

Ziqi Wan and Jie Wu
Department of Computer and Information Sciences
Temple University
Philadelphia, United States of America
{ziqi.wan, jiewu}@temple.edu

***Abstract*—The problem of Virtual Machine (VM) assignment in cloud computing is getting more and more attention each year. In this paper, we introduce the utility cost model, which combines both job time cost and the VM rent cost in the Cloud Scenarios. We investigate the interrelationship between the time cost and the VM rent cost, and formalize it as the parallel speedup pattern. We firstly propose the time first algorithm only considering the time cost, with the objective of minimizing the average completion time of all jobs. Additionally, we propose the price first algorithm, which focus on minimizing the rent cost. Based on that two algorithms, we introduce the policy shifting scheduling algorithm, which combines both time cost and rent price at the same time. We then formulate three group-based algorithms by adopting the idea of minimizing the utility cost. There are also time complexity and performance tradeoffs among the three group-based algorithms. Our experimental results demonstrate that our algorithms can achieve very good average utility gains in the real setting.**

***Keywords*—utility; virtual machine; assignment; parallel; cloud;**

I. Introduction

Cloud computing architectures have received increasing attention in recent years. Cloud providers take advantage of virtualization technologies to gain economic advantages from underutilized IT resources. Although the cloud data centers are increasingly larger, the number of jobs running in the cloud also grows explosively. Most of these jobs can be processed by multiple VMs in parallel. Thus, the VM resource is still limited. A good VM assignment strategy, which determines the number of VMs for each parallel job, is really needed.

A fundamental aspect for cloud providers is reducing data center costs while guaranteeing the promised Service Level Agreement (SLA) [1] to cloud consumers. Current virtualization technology offers the ability to easily relocate a virtual machine from one host to another without shutting it down, thus giving the opportunity to dynamically optimize the placement with a small impact on performance. Several recent works [2][3] addressed the VM assignment problem by minimizing the average finishing time of jobs assigned to machines, and several algorithms [4][5] have been proposed with the objective of maximizing the utilization of the virtual machines.

However, to the best of our knowledge, there is no proposal, explicitly considering both time cost and VM rent cost together. What's more, the average finishing time of the jobs and rent cost of VM are interrelated. However, most of the previous studies focus on either the execution time of the MapReduce jobs or the machine rent cost of the cloud providers, abiding by the distinct difference between them but ignoring the dependence therein.

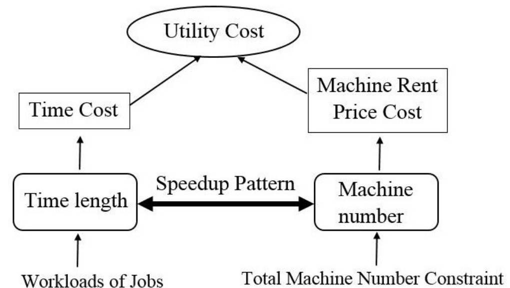


Fig. 1. Utility Cost Model

The problem of cost reduction becomes even more complex when considering a relationship between the processing speed of a job and the number of processing machines. Indeed, in cloud computing, the speedup pattern is not simply linear. CPU, memory and I/O resources will all influence the job processing speed. It becomes more challenging when a large number of jobs are competing for these resources. It is clear that only considering time cost or machine rent cost is not good enough. The utility of the MapReduce[6] jobs, which combines both jobs and machines, has not been carefully studied. What's more, due to the complexity of the speedup patterns of the real cloud clusters, the scheduling policies should either adapt to specific changing patterns or be robust.

This paper focuses on the utility-based virtual machine assignment issue in the MapReduce framework [6]. Usually, a typical MapReduce job includes both MapTasks and ReduceTasks. Each MapTask generates intermediate data in key-value pairs after taking a block of input data. Then, ReduceTasks fetch these intermediate data according to keys through the copy/shuffle phase, and proceed to the reduce phase after receiving all the intermediate results. The management of computing resources is done through the allocation of job slots; each slot only permits a single job to be launched. The number of machines for each job are not flexible for the Reduce phase. Thus, we focus on the Virtual Machine assignment problem of the map phase in this paper.

Our main contribution of this paper is inviting the Utility Cost Model as shown in Fig. 1, which combine both time cost and machine rent price cost. In the traditional model, only time cost or the machine cost was considered separately in the virtual machine placement issues. The rest of the paper is organized as follows. Firstly, we put our effort into the analysis of the parallel speedup patterns in the cloud clusters, then

discuss machine rent model and the utility model based on the basic idea of time cost and machine rent cost in Section II. In Section III, we provide several algorithms. We then perform real experiments in our MapReduce clusters. The results in Section IV answers the question about what the real speedup pattern is in the real cluster, and shows good evidence that our algorithm works well with the consideration of both time and price cost. In Section V, we list recent related works of others. Section VI concludes the paper and discusses possible future work.

II. Model Description

In this section, we will first illustrate the machine cost model, then explain the notion of utility, which clarifies the objective of our VM assignment strategy in the cloud scenarios.

A. Machine Rent Models

First of all, we take a look at the price model on the market. Amazon Web Services [7] provides a complete set of Cloud Computing services that enable one to build sophisticated, scalable applications. The basic idea is to pay only for what you use. In this paper, users pay for computation capacity by the hour with no long-term commitments or upfront payments. Users can increase or decrease their computation capacity depending on the demands of their applications and only pay the specified hourly rate for the instances they use. Thus, it is more flexible and interesting. What's more, most cloud provider consider every virtual machine in the same type to be the same, with the same computing capacity and unit rent price.

To better illustrate machine rent model, we define the notion of the total machine time of a job. We assume that the workload of job_i is w_i, and m_i is the number of machines for job_i. We assume all VMs have the same property, and each job is processed by multiple machines in parallel. Thus, we assume that a job starts and finishes at the same time for each VM running it. So they have the same processing time as $w_i/S(m_i)$ for job_i. Then the total machine time of job_i is $Mtime_i = w_i \times m_i/S(m_i)$. For the On-Demand Instances, the machine rent cost is linearly related to the total machine time. We define the rent price of virtual machines as p_i for job_i, a is a constant coefficient representing the price for renting each machine per unit time.

$$p_i = a \times Mtime_i = a \times w_i \times m_i/S(m_i) \tag{1}$$

B. Utility Model

The utility cost model is shown in Fig. 1. We extend the utility-based routing model from economics to cloud computing; we assume B_i is the benefit of completing a request, p_i is the service price, and t_i is the finishing time of job request i. We define the start time of job_i as t_i^{start}, so the processing time of job_i is $w_i/s(m_i)$. The relationship between the workload and the processing speed is $w_i = \int_{t_i^{start}}^{t_i} S(m_i(t))dt$. Once a job been launched, the number of machines for this job can not be changed any more in the Hadoop configuration. The finishing time of job_i can be transferred as follows.

$$t_i = w_i/s(m_i) + t_i^{start} \tag{2}$$

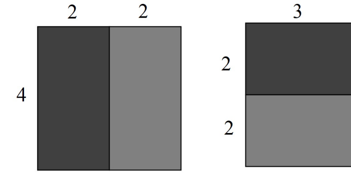


Fig. 2. Dominance Scheduling and Shared Scheduling

We first define $Uc_i = p_i + b \times t_i$ as the utility cost of job_i. Then we define the utility of job_i as $U_i = B_i - Uc_i = B_i - p_i - b \times t_i$. Here b is the coefficient representing the utility cost of a unit time of a job. It can be seen that the user is willing to give up response time in exchange for service price without any satisfaction change. Then we assume that the benefit of completing a request is proportional to the workload of a job. We define the coefficient B as the utility benefit per unit workload, then $B_i = B \times w_i$ as the benefit from finishing a request. Therefore, the utility function of a job request can be rewritten as,

$$\begin{aligned} U_i &= B_i - p_i - b \times t_i \\ &= Bw_i - aw_i m_i/S(m_i) - bw_i/S(m_i) - bt_i^{start} \\ &= w_i(B - am_i/S(m_i) - b/S(m_i)) - bt_i^{start} \end{aligned} \tag{3}$$

There are also constraints for machines and processing times. We assume that the number of machines M are limited. Therefore, there must be a performance tradeoff between price and time, when the processing speed is not linearly related to the number of machines. If one wants to finish earlier with more machines, he has to suffer a higher price. Our objective to maximize the overall utility of all users, which is $U = \sum U_i$.

When the processing speed is linearly increasing with the number of processing machines, the processing speed of a job with m machines is $S(m) = k \times m$. Then it becomes the malleable scheduling problem [8]. Therefore, the utility function can be rewritten as,

$$\begin{aligned} U_i &= Bw_i - akw_i m_i/S(m_i) - bt_i \\ &= Bw_i - akw_i m_i/km_i - bt_i \\ &= Bw_i - aw_i - bt_i = (B - a)w_i - bt_i \end{aligned} \tag{4}$$

From the equation above, we can see that the rent price for each job is fixed. To maximize the overall utility is equal to minimizing the average completion time.

In the sublinear speedup pattern and the superlinear speedup pattern, we assume $S(m) = k \times m \times \alpha^{m-1}$. Then the utility function becomes $U_i = Bw_i - akw_i m_i/S(m_i) - bt_i = Bw_i - akw_i m_i/(km_i\alpha^{m_i-1}) - bt_i = Bw_i - aw_i - bt_i = (B-a)w_i/\alpha^{m_i-1} - bt_i$. Here $\alpha \neq 1$, since the processing speed cannot linearly increase with the number of machines. Therefore, the utility of a job is not only determined by the job workload w_i and the finishing time t_i, but also by the speedup pattern α and the number of machines that have been used m_i.

III. Algorithms

Nowadays, most cloud service providers use the FIFO scheduler. There are other advanced schedulers like priority scheduler [9] and small job first scheduler, which show good

Algorithm 1 $TimeFirst$

Input: Workloads of all jobs, total number of machines, and speedup property of machines;
1: Compute t_d^{avg} and p_d for the dominance policy;
2: Compute t_s^{avg} and p_s for the shared policy;
3: **if** $t_d^{avg} = t_s^{avg}$ **then**
4: **if** $p_d = p_s$ **then**
5: Apply tie-breaking rules to find a better policy;
6: **else**
7: The policy with the lower p is better;
8: **else**
9: The policy with the lower t is better;
10: Schedule jobs according to the better policy.

Algorithm 2 $PriceFirst$

Input: Workloads of all jobs, total number of machines, and speedup property of machines;
1: Compute t_d^{avg} and p_d for the dominance policy;
2: Compute t_s^{avg} and p_s for the shared policy;
3: **if** $p_d = p_s$ **then**
4: **if** $t_d^{avg} = t_s^{avg}$ **then**
5: Apply tie-breaking rules to find a better policy;
6: **else**
7: The policy with the lower t is better;
8: **else**
9: The policy with the lower p is better;
10: Schedule jobs according to the better policy.

performance in minimizing the average completion time of jobs. However, these schedulers all adopt the dominance scheduling policy, which allocates a job with as many slots as possible to greedily achieve an early finishing time for each job. Actually, sometimes the shared scheduling is equal to or better than the dominance scheduling policy. For example, in Fig. 2, there are 2 jobs; assume the processing time for each task with 4 slots is 2, and the the processing time for each task with 2 slots is 3. Then the average completion times of different policies are equal in this example. The shared policy has a shorter machine time for each job, but it has a lower utility cost than the other. In this section, we provide several algorithms trying to maximize the overall utility. We first consider two special cases, in which either time cost or rent price is more important than the other.

A. Time First and Price First

In some cases, users only care about the time, and pay little attention to the rent price. Therefore, we should minimize the time cost first, then consider minimizing the machine rent price.

Although we try to minimize each job's completion time, it might generate a very large overall completion time. Here we assume $w_1 \leq w_2 \leq ... \leq w_n$ are the workloads of n jobs in a batch, and $n \leq M$. Our policy still follows the idea of the shortest workload job first rule. We define $t_{d1}, t_{d2}, ..., t_{dn}$ as the finishing times of n jobs for the dominance policy. The average finishing time for the dominance scheduling policy is,

$$\begin{aligned} t_d^{avg} &= (t_{d1} + (t_{d1} + t_{d2}) + ... + (t_{d1} + t_{d2} + ... + t_{dn}))/n \\ &= (nw_1 + (n-1)w_2 + ... + w_n)/(nS(M)) \end{aligned} \quad (5)$$

The average finishing time for the shared scheduling policy is shown as follow.

$$t_s^{avg} = \sum w_i/(n \times S(\frac{M}{n})) \quad (6)$$

When a tie-breaking decision is needed, we need to consider the rent machine price as well. There are some other tie breaking rules, such as uniform random selection; this considers the rent cost of machines of the dominance policy.

$$p_d = \frac{M \times S(1) \times \sum w_i}{S(M)} \quad (7)$$

The rent cost of machines of the shared policy is shown as follow.

$$p_s = \frac{M \times S(1) \times \sum w_i}{n \times S(M/n)} \quad (8)$$

Our idea to try and find out the changing point for the dominance scheduling policy and shared scheduling policy. Our time first algorithm are shown in Alg. 1. In the worst case, we might always failed to choose the better policy; however, there is a bound for the average finishing time.

Theorem 1. *In the worst case, the average finishing time is* $max(2n/(n+1), S(\frac{M}{n})/S(M))$ *times the optimal one.*

Proof: In the worst case, we choose the wrong scheduling policy due to the lack of future knowledge. The ratio of average makespan of two scheduling algorithms is $t_d^{avg}/t_s^{avg} = \frac{(nw_1+(n-1)w_2+...+w_n)/(nS(M))}{\sum w_i/(n*S(\frac{M}{n}))}$. If we always choose the sharing schedule policy, in the worst case, $w_1 = w_2 = ... = w_n$ and $S(\frac{M}{n}) = S(M)/n$, then $t_d^{avg}/t_s^{avg} = \frac{(n+1)S(\frac{M}{n})}{2S(M)} = (n+1)/2n$, and the bound is $2n/(n+1)$. If we always choose the dominance schedule policy, then the worst case is that w_1 is much greater than the $w_2, ..., w_n$, then $t_d^{avg}/t_s^{avg} = \frac{w_1}{S(M)}/\frac{w_1}{S(\frac{M}{n})} = S(\frac{M}{n})/S(M)$. ■

In the case of users only caring about the price and paying little attention to the time cost, we should minimize the price first, and consider the time cost latter. Thus, we assign the minimum number of machines for each job for the sublinear speedup pattern. Our price first algorithm are shown in Alg. 2. The best idea is to maximize the number of machines for each job.

B. Policy Shifting Scheduling

As we known, the price first algorithm and the time first algorithm have failed to combine both rent price and time cost together. Sometimes, we need to decide whether we should change the scheduling rules by monitoring the number of job requests in the cloud and their remaining workloads. The Policy Shifting Scheduling algorithm are shown in Alg. 3. In the worst case, we might always failed to choose the better policy; fortunately, we figure out a bound for the average utility.

Algorithm 3 $PolicyShifting$

Input: Workloads of all jobs, total number of machines, and speedup property of machines;

1: Compute t_d^{avg} and p_d for dominance scheduling policy;
2: Compute t_s^{avg} and p_s for shared scheduling policy;
3: Compute and compare the utility cost of two policies;
4: Schedule jobs according to the better policy with lower utility cost.

Algorithm 4 $Group\ Utility - single\ size$

Input: Workloads of all jobs, total number of machines, and speedup property of machines;

1: **for** g from 1 to maximum number of machines M **do**
2: Set $g = \lfloor N/M \rfloor$ as the number of jobs in all groups, except the last group. The number of jobs in the last group is $g_{last} = N - g$;
3: Group all the jobs in their workloads. The job smaller workload arranged in the earlier processing group;
4: Compute total $U = \sum(B_i - Uc_i)$ of the all group;
5: Compare and find out the best number of jobs for each group. Schedule jobs in groups with that number.

Algorithm 5 $Group\ Utility - all\ sizes$

Input: Workloads of all jobs, total number of machines, and speedup property of machines;

1: Set the number of jobs in group j as g_j, and J as the total number of groups.
2: Set total the number of jobs in groups as $N = \sum g_j$.
3: **for** Each possible set of $g_1, g_2, ..., g_N$ **do**
4: Group all the jobs in their workloads. The job having smaller workload is arranged in the earlier processing group;
5: Compute total $U = \sum(B_i - Uc_i)$ of the all group;
6: Compare and find out the best number of jobs for each group. Schedule jobs in groups with that number.

Theorem 2. *In the worst case, the utility cost is* $a \times max(2n/(n+1), S(\frac{M}{n})/S(M)) + b \times S(\frac{M}{n})/S(M)$ *times the optimal one.*

Proof: In the worst case, we suffer both worst time cost and worst rent price of the two algorithms at the same time. According to theorem 1, we have the average finishing time as $max(2n/(n+1), S(\frac{M}{n})/S(M))$. At the same time, the maximum price is $S(\frac{M}{n})/S(M)$. So the worst utility cost is $a \times max(2n/(n+1), S(\frac{M}{n})/S(M)) + b \times S(\frac{M}{n})/S(M)$. ■

C. Utility-based Scheduling

The intuition of this policy is to find the proper number of machines for each job with the maximal utility. Jobs should follow the order of the smallest workload first policy then determine the processing sequence. In order to maximize the utility, we need to minimize the total utility cost Uc_{total}. For job_i we assume that the number of machines used does not changed during the processing, and every machine is fully utilized. As we known $Uc_i = p_i + b \times t_i$, our objective function can be written as follows.

$$Uc_{total} = \sum_i^N Uc_i = \sum_i^N (p_i + b \times t_i) \tag{9}$$

Here, we use the overfitting function of the processing speed to find the best m_i. As $S(m_i) = k \times m_i \times \alpha^{m_i - 1}$, the total cost of job_i is

$$\begin{aligned} Uc_{total} &= \sum_i^n (w_i(am_i + b)/S(m_i) + bt_i^{start}) \\ &= \sum_i^n (w_i(am_i + b)/(km_i\alpha^{m_i-1}) + bt_i^{start}) \end{aligned} \tag{10}$$

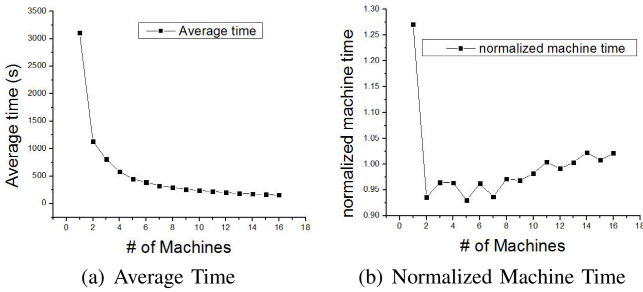
A naive way is to schedule jobs one by one. Each job is greedily allocated the optimal number of machines. The computation complexity is *O(N)*. Here, N is the total number of jobs. However, this algorithm is actually too bad to use. It is often the case that one job will use up all the machines at a time. The waiting time for the following machines will aggregate. However, the greedy algorithm of scheduling jobs one by one can be easily extended to the scheduling policy for a group of jobs to get better a performance with the sacrifice of time complexity of the algorithm. We assume that the number of jobs in a group j is g_j, and $n \leq M$. The total number of groups is J. It is clear that $\sum_j^J g_j = N$.

Algorithm 6 $Group\ Utility - greedy\ size$

Input: Workloads of all jobs, total number of machines, and speedup property of machines;

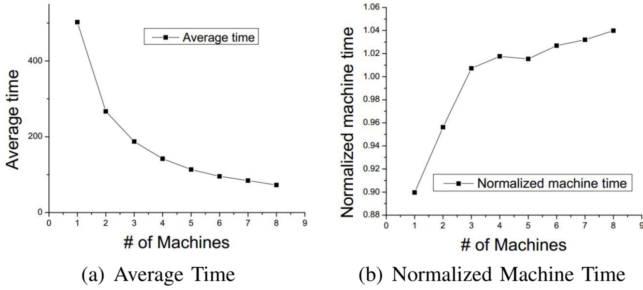
1: h=1
2: **while** There is a jobs not in groups $g_1, g_2, ..., g_h$ **do**
3: **for** g_h from 1 to maximum number of machines M **do**
4: **while** $N > \sum_1^h g_j$ **do**
5: Set Number of jobs in g_{h+1} as g_h
6: Compute total $U = \sum(B_i - Uc_i)$ of the all group;
7: $h = h + 1$;
8: Compare and find out the best number of jobs for g_h. Schedule jobs in groups with that number.

Here we provide three group utility algorithms. The first one is called single size algorithm as shown in Alg. 4. In this algorithm, we assume all groups have the same size, which means all groups contains the same number of jobs. The complexity of this algorithm is *O(MN)*, which is quite small. But the restriction of having the same size for each group is too strong The second group utility algorithm is the all sizes algorithm as shown in Alg. 5. In contrast to the single size algorithm, it considers all the possible group sizes. Although it may get a very good result, it is very time consuming to enumerate all the possible combinations of group sizes. The worst time complexity of this algorithm is *O($M!^N$)*. Since both the single size algorithm and the all sizes algorithm have some obvious drawbacks, we provide the greedy size algorithm to make a balance between performance and time complexity. The idea of the greedy size algorithm is to greedily determine the number of jobs in each group as shown in Alg. 6. The worst time complexity of this algorithm is *O(M^N)*.



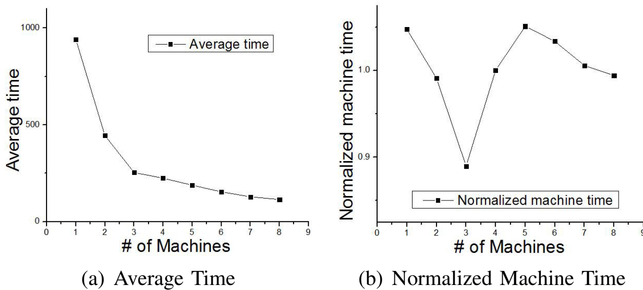
(a) Average Time (b) Normalized Machine Time

Fig. 3. Word Count



(a) Average Time (b) Normalized Machine Time

Fig. 4. Pentomino



(a) Average Time (b) Normalized Machine Time

Fig. 5. TeraSort

Theorem 3. *The utility cost of based on the single size algorithm utility group is less than* $(a+b) \times max_n(S(\frac{M}{n})/S(M))$ *times of the optimal one.*

Proof: In the utility group algorithm, we try to maximize the total utility of the group. However, the number of jobs in a group is fixed before the scheduling process. However, the number of jobs in a group will influence the scheduling flexibility. In the worst case, there are M jobs for M machines. The utility cost in this situation is $(a+b)max_n(S(\frac{M}{n})/S(M)) \times OPT$. ■

In general, the greedy size policy gets better results than the single size policy, and gets results close to those of the all size policy. However, the number of jobs in a group needs to be further discussed, and what we discussed here is in the batched start offline situation. For the online condition, more study need to be done. It might be our future work.

IV. Experiments

We investigate our model based on the real experiment. Our clusters consist of Dell R210 Servers. Each server has a dual core Intel Celeron processor. They also have 4 GB of RAM. They then have Gigabit Ethernet jacks for communication. We also use a Cisco small business 300 Series Managed Switch to connect the servers together. Each Switch has only eight ports available so only eight servers can be connected through one hop. To this end, we will collect data by adding servers one at a time and running the same job with the same input data ten times on each configuration. We will stop after we have done this with the first sixteenth servers. From there we will try to estimate how time will be affected by adding more nodes.

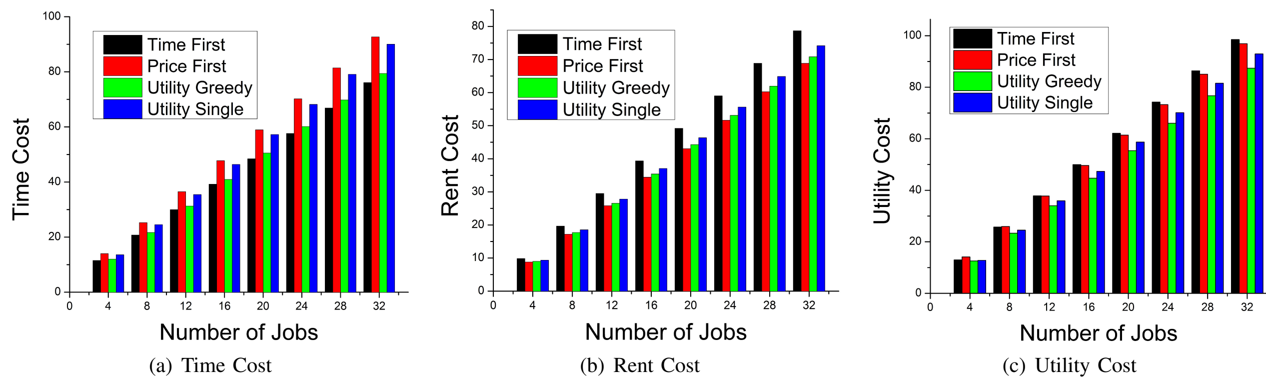
In this paper, we investigate three common applications in the Hadoop cloud framework, which are Word Count, Pentomino [10], and TeraSort [11]. In the following paragraph, we will present our results and point out some interesting things to note about them. Figs. 3(a), 4(a), 5(a) show the relationship between the average finishing time and the parallel machine numbers of the three cloud applications. It is clear that increasing the number of processing machines for any type of job will accelerate the processing speed and will bring on the shorter finish time of the job. However, the speedup patterns are no similar. The processing speed for a Word Count job increases rapidly only from 1 VM to 2 VMs and slows down when adding more machines. However, the processing speed of the Pentomino job keeps boosting. For the TeraSort job, the processing speed rises significantly then slows down drastically.

Figs. 3(b), 4(b), 5(b) illustrate the relationship between the normalized machine time and machines number of the three cloud applications. Here the normalized machine time (NMT) is the normalized product of the parallel processing time $t_i - t_i^{start}$ and the number of parallel processing machines m, namely $NMT = \eta(t_i - t_i^{start}) \times m$. Here η is the normalized coefficient. The normalized machine time is used to measure the overall machine usage, and is in proportion to the machine rent cost. The higher NMT, the higher the machine rent cost. From the above observation, we can see that, no matter how different those NMT patterns are, the changing points do exist. It may help us to find out the best choice of machine numbers.

Then we test our algorithms by using the results we get from the real clusters. The time costs of 4 algorithms are shown in Fig. 6(a). It is clear that the price first policy has a very large time cost compared to other policies, and the time first policy has the lowest time cost. The reason is that the price first policy always uses a few machines with minimal cost for every job, so there are many machines unassigned when the number of jobs is very small. The machine rent costs of 4 algorithms are shown in Fig. 6(b). In contrast to the result of the time cost, the time first policy has the worst machine rent price. The reason is that the time first policy always uses as many machines as possible. Due to the sub-linear increase of the parallel computing speed, the overall machine rent time increased. However, when combining both time and rent cost, neither the price first policy nor the time first policy are the best as shown in Fig. 6(c). As a result, the utility group policies are more powerful in minimizing the utility cost of the cloud clusters. We can also see that the greedy step group policy is better than the single step policy.

V. Related Work

There is a large body of research work on the performance optimization for the Cloud. We list the closely related work



(a) Time Cost (b) Rent Cost (c) Utility Cost

Fig. 6. Simulation Results of 4 Algorithms for Word Count

to ours as follows. Many MapReduce schedulers have been proposed over the past few years trying to maximize the resource utilization. Zaharia et al. introduced delay scheduling [12] that speculatively postpones the scheduling of the head-of-line jobs and ameliorate the locality degradation in Hadoop Fair scheduler. They also proposed Longest Approximate Time to End [13] scheduling policy to mitigate the deficiency of Hadoop scheduler in coping with the heterogeneity across virtual machines in a cloud environment.

Some recent publication addressed the VM assignment problem by minimizing the average finishing time of jobs assigned to machines. Wang et al. [2] studied the m identical parallel-machine and unrelated parallel-machine scheduling with a deteriorating maintenance activity to minimize the total completion time. Albers et al. [3] investigated dynamic speed scaling, a technique to reduce energy consumption in variable-speed microprocessors.

Several algorithms have been proposed with the objective of maximizing the utilization of the virtual machines. For example, Beloglazov et al. [4] [5] proposed approaches for known stationary workload and a given state configuration optimally solves the problem of host overload detection by maximizing the mean intermigration time under a Markov chain model.

Recently, YARN [14] has been proposed by Yahoo! as the next generation MapReduce. It separates the JobTracker into ResourceManager and Application Manager, and removes task slot concept. In future, we plan to incorporate our techniques into the YARN.

VI. Conclusion

We consider the design and analysis utility-based scheduler in the cloud environment. Unlike all existing works, we propose the notion of the utility for the Virtual Machine management. Next, we investigate the parallel speedup pattern in the cloud clusters. After that, we propose several scheduling algorithms based on the idea of the time cost and rent price. Then we introduce the policy shifting scheduling algorithm, provided with bounded performance against the optimal one. Motivated by the previous scheduling policies, we provide the three utility-based algorithms, with each method having its unique pros and cons. Our experimental results demonstrate that our algorithms can achieve very good average utility in the given settings. The model presented here opens the door for an in-depth study of how to schedule in the presence of phase overlapping. There are a wide variety of open questions remained to be done.

References

[1] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1–11, 2011.

[2] L.-Y. Wang, X. Huang, P. Ji, and E.-M. Feng, "Unrelated parallel-machine scheduling with deteriorating maintenance activities to minimize the total completion time," *Optimization Letters*, vol. 8, no. 1, pp. 129–134, 2014.

[3] S. Albers, F. Müller, and S. Schmelzer, "Speed scaling on parallel processors," *Algorithmica*, vol. 68, no. 2, pp. 404–425, 2014.

[4] A. Beloglazov and R. Buyya, "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 7, pp. 1366–1379, 2013.

[5] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 159–168, IEEE, 2010.

[8] W. Ludwig and P. Tiwari, "Scheduling malleable and nonmalleable parallel tasks," in *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pp. 167–176, Society for Industrial and Applied Mathematics, 1994.

[9] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Job scheduling strategies for parallel processing*, pp. 110–131, Springer, 2010.

[10] A. Hadoop, "Hadoop," 2009.

[11] O. OMalley and A. C. Murthy, "Winning a 60 second dash with a yellow elephant," *Proceedings of sort benchmark*, 2009.

[12] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, pp. 265–278, ACM, 2010.

[13] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments.," in *OSDI*, vol. 8, p. 7, 2008.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.