Utility-based Uploading Strategy in Cloud Scenarios

Ziqi Wan, Jie Wu and Huanyang Zheng
Department of Computer and Information Sciences, Temple University, USA
Email: {ziqi.wan, jiewu, huanyang.zheng}@temple.edu

***Abstract*—There is a great potential to boost the performance of mobile devices by offloading computation-intensive parts of mobile applications to the cloud. However, this potential is hindered by a gap between how individual mobile devices demand computational resources and how cloud providers offer them: offloading requests from a mobile device usually require a quick response, which may be infrequent, and is subject to variable network connectivity, whereas cloud resources incur relatively long setup times, are leased for long time quanta, and are indifferent to network connectivity. In this paper, we present the design of utility-based uploads sharing strategy in cloud scenarios, which bridges the above gap through providing computation offloading as a service to mobile devices. Our scheme efficiently manages cloud resources for offloading requests to improve offloading performances of mobile devices, as well as to reduce the monetary cost per request of the provider. We also schedule offloading requests to resolve the contention problem for cloud resources. The proposed scheme makes offloading decisions with a controlled risk to overcome the uncertainties caused by variable network connectivity and program execution. Simulation results show that the proposed scheme can reduce the costs of cloud resources and enable mobile computation speedup for mobile devices.**

Keywords*—*Computation, connectivity, mobile devices, offloading, resource management.

I. Introduction

The idea of offloading computation from mobile devices to remote computational resources to improve performance and reduce energy consumption has been addressed over the past decade. The usefulness of computation offloading hinges on the ability to achieve high computation speedups with small communication delays. In recent years, this idea has received more attention due to the significant rise of mobile applications, the availability of powerful clouds, and the improved connectivity options for mobile devices. By identifying the offloadable tasks at runtime, recent work has aimed to generalize this approach to benefit more mobile applications.

Despite great potential, a key challenge in computation offloading lies in the mismatch between how individual mobile devices demand and access computational resources and how cloud providers offer them. Offloading requests from a mobile device require quick responses which may be infrequent. Therefore, the ideal computational resources suitable for computation offloading should be immediately available upon request and should be quickly released after execution. In contrast, cloud computational resources have long setup times and are leased for long time quanta. For example, it takes about 27 seconds to start an Amazon EC2 VM instance. The time quantum for leasing an EC2 VM instance is one hour. If an instance is used for less than an hour, the user must still pay for one-hour usage. This mismatch can thus hamper offloading performance and incur high monetary cost.

Another challenge 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. 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. We need to monitor the utility decay speed and waiting benefit for sharing. The longer the time one waits, the higher the time cost, but the probability of sharing VM with more users is also higher. In this case, our main focus is balancing the time cost and sharing benefit. It is clear that only considering time cost or machine rent cost is not good enough. The utility of the MapReduce jobs, which combines both job makespan and type of machine, has not been carefully studied. As shown in Fig. 1, due to the complexity of the speedup patterns of the real cloud clusters, the scheduling policies should either adapt to the changing pattern or be more robust.

In this paper, we propose a strategy for users to share the cloud services to boost their performance by speeding up their process while minimizing their rent cost from the cloud machines. A system that bridges the above-discussed gaps by providing computation offloading as a service provide an intermediate service between a commercial cloud provider and mobile devices can make the properties of underlying computing and communication resources transparent to mobile devices and can reorganize these resources in a cost-effective manner to satisfy offloading demands from mobile devices. An uploading management system receives mobile user computation offload demands and allocates them to a shared set of compute resources that it dynamically acquires (through leases) from a commercial cloud service provider. The goal of this system is to provide the benefit of computation offloading to mobile devices while, at the same time, minimizing the leasing cost of computational resources.

Our main contributions are summarized as follows:

- We introduce a utility model, which combine both time cost and machine rent price cost.
- Multiple efficient algorithms are proposed to maximize the utility gains in the cloud settings. The optimality prerequisite is explored. The performances of the proposed algorithms are analytically studied.

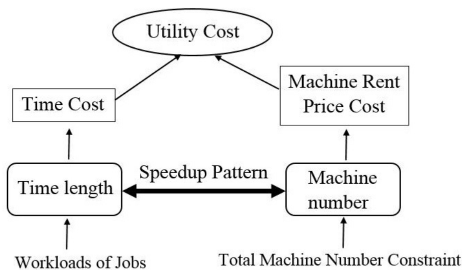


Fig. 1. Utility model.

- Extensive experiments are conducted to evaluate the proposed solutions. The results are shown from different perspectives to provide insightful conclusions.

We start by formulating an optimization problem whose solution can guide the required decision making in Section 2. In Section 3, we develop a set of novel techniques, including resource-management mechanisms that select resources suitable for computation offloading and adaptively maintain computational resources according to offloading requests. In Section 4, we propose several greedy algorithms that properly allocate offloading tasks to the cloud resources with limited control overhead. The results in Section 5 shows good evidence that our algorithm works well in practice. In Section 6, we list recent related works of others. Section 7 concludes the paper and discusses possible future work.

II. Background and Problem Statement

A. Background

Cloud computation resources are usually provided in the form of virtual machine (VM) instances. To use a VM instance, a user installs an OS on the VM and starts it up, both incurring delay. VM instances are leased based on a time quanta. Amazon EC2 uses a one-hour lease granularity. If a VM instance is used for less than the time quanta, the user still needs to pay for that usage. A cloud provider typically provides various types of VM instances with different properties and prices. We provide some properties and prices for three types of Amazon EC2 VM instances: Standard On-Demand Small instance (m1.small), Standard On-Demand Medium instance (m1.medium) and High-CPU On-Demand Medium instance (c1.medium). For some pricing models (e.g., EC2 spot), the leasing price may change over time. Note that the server component of offloaded mobile computation needs to run on a VM instance. This server component needs to be launched at the time the offloading request is made, and terminated when the required computation is complete. The lifetime of the server component is typically much less than the lease quantum used by the cloud service provider. An important question we consider in our system design is how to ensure that there is enough VM capacity available to handle the mobile computation load without needing to always launch VM instances on-demand and incur a long setup time.

B. Problem Statement

A basic computation-offloading system is composed of a client component running on the mobile device and a server component running in the cloud. The client component has

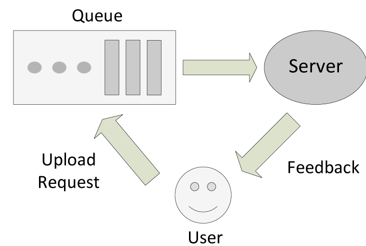


Fig. 2. A waiting buffer.

three major functions. First, it monitors and predicts the network performance of the mobile device; Second, it tracks and predicts the execution requirements of mobile applications in terms of input/output data requirements and execution time on both the mobile device and the cloud; Third, using this information the client component chooses some portions of the computation to execute in the cloud so that the total execution time is minimized. The server component executes these offloaded portions immediately after receiving them and returns the results back to the client component so that the application can be resumed on the mobile device. Computation offloading trades off communication cost for computation gain. Previous systems usually assume stable network connectivity and adequate cloud computation resources. However, in mobile environments a mobile device may experience varying or even intermittent connectivity, while cloud resources may be temporarily unavailable or occupied. Thus, the communication cost may be higher, while the computation gain will be lower. Moreover, the network and execution prediction may be inaccurate, causing system performance degradations.

Our basic idea is manipulating a waiting queue between mobile device users and cloud server to achieve good offloading performance at low monetary cost by sharing cloud resources among users, as shown in Fig. 2. Specifically, the goal is to minimize the usage cost of cloud resources under the constraint that the speedup of using cloud service against local execution. The problem of offloading can be further divided into the following two subproblems:

- Cloud resource management: This is the problem of determining the number and type of VM instances to lease over time. It has two major goals. First, there should always be enough VM instances to ensure high offloading speedup. Second, the cost of leasing VM instances should be minimized.
- Offloading decision: This is the problem of deciding whether a mobile device offloads a computation task. The challenge comes from the uncertainties of network connectivity, program execution, and resource contention. A wrong offloading decision will both waste cloud resources and result in lower speedup. It is very important to properly handle the uncertainties.

III. Model Description

A. Offloading Gain and Risk

When an offloadable task, O_k, is initiated at time t_k, the offloading controller needs to determine if it is beneficial to offload this task to the cloud. Let us use T_{ws} to denote the

time to wait for connectivity before sending the data, T_s for the time to send the data, T_c for the execution time of the task in the cloud, T_{wr} for the time to wait for connectivity before receiving the result, and T_r for the time to receive the result. The local execution time is $L(O_k)$, which is estimated by the execution predictor. The response time of offloading to an active Server, $R(O_k)$, can be expressed as: $R(O_k) = T_{ws} + T_s + T_c + T_{wr} + T_r$. It is beneficial to offload only if the local execution time is longer than the response time of offloading. Therefore, we use their difference to represent the offloading gain: $G = L(O_k) - R(O_k)$. Because of the uncertainties in the mobile environment, the offloading controller can only obtain a distribution for G (i.e., $E(G)$). Simply using $E(G)$ to make the offloading decision will introduce the risk of longer execution time.

Our risk-controlled offloading is based on two key ideas. First, we use risk-adjusted return in making the offloading decision so that the return and risk of offloading are simultaneously considered. Specifically, $E(G)$ and $\sigma(G)$ are used as the return and risk of the offloading gain, respectively. Thus, the risk-adjusted return of offloading gain is $E(G) - \sigma(G)$. When its value is larger than a certain threshold, the computation task will be offloaded to the cloud. Otherwise, it will be locally executed. Second, we re-evaluate the return and risk when new information is available. When a computation task is initiated, the offloading controller evaluates its return and risk of offloading gain. The detailed algorithms to compute them are described in the appendix. If the risk-adjusted return (i.e., $E(G)$, $\sigma(G)$) is larger than a threshold, the offloading controller offloads the task to the cloud. In addition, it also listens to the connectivity status which has a high impact on $E(G)$ and $\sigma(G)$. Once new connectivity information is updated, it re-evaluates the risk-adjusted return and adjusts its decision accordingly.

B. Utility-based Offloading Model

We extend the utility-based model from economics to cloud computing; we assume A is the award of completing a request, R is the service price, C is the utility decay based on time cost. We first define $U = A - R - C$ as the utility of a job. We assume the rent cost is equally shared by each job in the cloud at a certain time period. Let n_t be the number of jobs in the cloud at time t. Thus the highest possible rent cost at time t is $R_t = \alpha/n_t$. α is the total rent cost of the cloud service during a certain time period. The number of jobs in a certain time period starting with t_0 can only increase as the number of jobs keep growing, then we have $n_{t_s} = n_{t_0} + \Delta n_{t_1} + \Delta n_{t_2} + ... + \Delta n_{t_f}$. t_s is the job start time, Δn_{t_i} is the number of newly added jobs on time slot t_i.

1) Linear time Model: We first assume the utility decays linearly with time. Then $C = \beta(t_f - t_a)$, β is the unit time decay, t_a is the job arrival time, and t_f is the finishing time of a job request.

$$U_t = A - \alpha/n_{t_s} - \beta(t_f - t_a) \tag{1}$$

Note that the job start time t_s is not necessary to be the job request arrival time. Therefore, there is a tradeoff between rent cost and time decay. For example, when a job request has been generated, it can choose to wait for more jobs coming to lower the rent cost or to get into the cloud as soon as possible to minimize the time decay. It would be more complicated, if a deadline or a service level agreement is concerned. To make it simple, we cut the time line into discrete time slots. It is also practical, as the cloud service takes job requests in each time slot to avoid large communication overhead. Assume the utility gains of a job starting at time slot i and time slot j ($j > i$) are U_i and U_j. The time length of a slot is Δt.

$$\begin{aligned} U_i - U_j &= -\alpha(1/n_i - 1/n_j) - \beta(t_i - t_j) \\ &= \alpha(1/n_j - 1/n_i) + \beta\Delta t(j - i) \\ &= -\alpha(\Delta n_{i+1} + ... + \Delta n_j)/n_j n_i + \beta\Delta t(j - i) \end{aligned} \tag{2}$$

Let $Y = \Delta n_{i+1} + \Delta n_{i+2} + ... + \Delta n_j$, the above equation can be simplified as $U_i - U_j = -\alpha Y/(n_i + Y)n_i + \beta\Delta t(j - i)$. We further assume the average growth of jobs in the cloud is Δn. Thus, $Y = (j-i)\Delta n$ and $U_i - U_j = (\beta\Delta t - \alpha\Delta n/(n_i + Y)n_i)(j-i)$. Now, we can see that if $\beta\Delta t - \alpha\Delta n/(n_i + Y)n_i$ greater than 0, the utility gain at time slot i is greater than the utility gain at time slot j. Especially, when $j = i + 1$, we can have $U_i - U_{i+1} = (\beta\Delta t - \alpha\Delta n/(n_i + \Delta n)n_i)$. So it is only based on the number of jobs n_i in the cloud at t_i. Further, we consider the choice of local process.

C. Offloading Decision

Complicating this issue is the fact that mobile devices access cloud resources over wireless networks which have variable performance and/or high service cost. For example, 3G networks have relatively low bandwidth, causing long communication delays for computation offloading. On the other hand, although WiFi networks have high bandwidth and are free to use in many cases, their coverage is limited, resulting in intermittent connectivity to the cloud and highly variable access quality even when connectivity exists. Therefore, the decision of maintaining the length of request queue, as shown in Fig. 3, is very important.

Assume the local processing speed is sl, and the cloud processing speed is sc. Then the acceleration of the cloud is $ac = sc/sl$. Adopt the utility function above. The local process has the utility of $U_{lt} = A - \beta(t_f - t_a)$. It does not have the shard rent cost of cloud. However, it is still possible that the local process has a higher utility cost. Here we assume that the workload of a job w is the same for cloud process and local process. Then the utility of local process is $U_{lt} = A - \beta w/sl$, and the utility of cloud process is $U_{ct} = A - \alpha/n_{ts} - \beta(w/sc + t_s - t_a)$. $U_{ct} - U_{lt} = \beta(w/sc - w/sl + t_s - ta) + \alpha/n_{t_s}$. It should be noted that there are many time cost models. However, the common thing among most of the time models is that the longer the time makespan is, the higher the cost.

Since job_i may not be the first job to start, we define the start time of job_i as t_i^{start}, so the processing time of of job_i is $w_i/s(m_i)$. The relationship between the workload and the processing speed is $w_i = \int_{t_i^{start}}^{t_i} S(m_i(t))dt$. However, the number of machines cannot be changed in the Hadoop configuration. The finishing time of job_i can be transferred as follows. The above is discussed under the assumption that cloud service is powerful enough. So there is not congestion in the cloud when the number of applications are huge. The key idea in deciding whether to wait for a cheap price or to start early to save money the decreasing speed of R and time decay; what's more, cloud processing should generate more

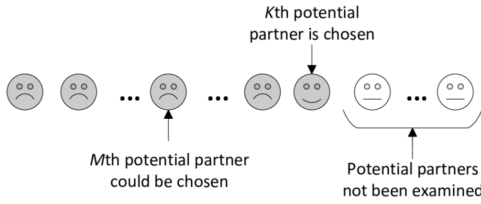


Fig. 3. Decide the number of requests in the queue.

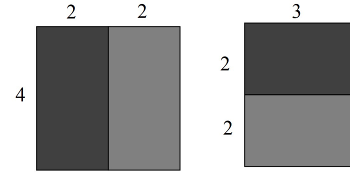


Fig. 4. Dominance Scheduling and Shared Scheduling.

utility gain than the local processing, otherwise we should adopt local process which is cheap and steady.

D. Offloading Decision under Monotonic Arrival

The offloading controller uses the information from the connectivity and execution predictors to estimate the potential benefits of the offloading service. Ideally, if future connectivity and execution times can be accurately predicted immediately after the mobile application starts, the offloading controller can make the global optimal offloading decision.

However, such global optimum is unavailable in reality. A good estimation will help a lot. Instead, the offloading controller uses a greedy strategy to make the offloading decision. A Markov chain (discrete-time Markov chain or DTMC) is a mathematical system that undergoes transitions from one state to another on a state space. It is a random process usually characterized as memoryless: the next state depends only on the current state and not on the sequence of events that preceded it. This specific kind of "memorylessness" is called the Markov property. Markov chains have many applications as statistical models of real-world processes.

A Markov chain is a sequence of random variables X_1, X_2, X_3, ... with the Markov property, namely that, given the present state, the future and past states are independent. Formally, $Pr(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = Pr(X_{n+1} = x \mid X_n = x_n)$, if both conditional probabilities are well defined, i.e. if $Pr(X_1 = x_1, ..., X_n = x_n) > 0$. The possible values of X_i form a countable set S called the state space of the chain. Markov chains are often described by a sequence of directed graphs, where the edges of graph n are labeled by the probabilities of going from one state at time n to the other states at time $n+1$, $Pr(X_{n+1} = x \mid X_n = x_n)$. The same information is represented by the transition matrix from time n to time $n+1$. However, Markov chains are frequently assumed to be time-homogeneous (see variations below), in which case the graph and matrix are independent of n and so are not presented as sequences.

These descriptions highlight the structure of the Markov chain that is independent of the initial distribution $Pr(X_1 = x_1)$. When time-homogenous, the chain can be interpreted as a state machine assigning a probability of hopping from each vertex or state to an adjacent one. The probability $Pr(X_n = x|X_1 = x_1)$ of the machine's state can be analyzed as the statistical behavior of the machine with an element x_1 of the state space as input, or as the behavior of the machine with the initial distribution $Pr(X_1 = y) = [x_1 = y]$ of states as input, where $[P]$ is the Iverson bracket. The stipulation that not all sequences of states must have a nonzero probability of occurring allows the graph to have multiple connected components, suppressing edges encoding a zero transition probability, as if *a* has a nonzero probability of going to *b* but *a* and *x* lie in different connected components, then $Pr(X_{n+1} = b|X_n = a)$ is defined, while $Pr(X_{n+1} = b|X_1 = x, ..., X_n = a)$ is not.

It would be more complicated if the decay function is more not linear. Another common decay function is the discounted time decay. In the discounted time decay model, the utility can be presented as $U = (A - R)^{\beta t} = (A - \alpha/n_{t_s})^{\beta(t_f - t_s)}$.

Theorem 1. *In a finite horizon monotone stopping rule problem, the one-stage look-ahead rule is optimal.*

Proof: Suppose the horizon is J . One optimal rule is

$$N^* = min(n \geq 0 : Y_n \geq E(V_{n+1}^J|F_n)) \tag{3}$$

where $V_{J+1}^J = \infty$, $V_J^J = Y_J$, and by backward induction

$$V_n^J = max(Y_n, E(V_{n+1}^J|F_n)), n = 0, 1, ..., J-1 \tag{4}$$

$$\begin{aligned} &Y_{J-1} \geq E(V_J|F_{J-1}) = E(V_J^J|F_{J-1}); \\ &Y_{J-2} \geq E(V_{J-1}|F_{J-2}) = E(V_{J-1}^J|F_{J-2}); \\ &... \\ &Y_n \geq E(V_{n+1}|F_n) = E(V_{n+1}^J|F_N) \end{aligned} \tag{5}$$

Thus, we can conclude that the one-stage look-ahead rule is optimal, in a finite horizon monotone stopping rule problem. ■

E. Offloading Decision under Random Arrival

In mathematics, the theory of optimal stopping is concerned with the problem of choosing a time to take a particular action, in order to maximize an expected reward or minimize an expected cost. Optimal stopping problems can be found in areas of statistics, economics, and mathematical finance (related to the pricing of American options). A key example of an optimal stopping problem is the secretary problem. Optimal stopping problems can often be written in the form of a Bellman equation, and are therefore often solved using dynamic programming.

The secretary problem is one of many names for a famous problem of the optimal stopping theory. The problem has been studied extensively in the fields of applied probability, statistics, and decision theory. It is also known as the marriage problem, the sultan's dowry problem, the fussy suitor problem, the googol game, and the best choice problem. The basic form of the problem is the following: imagine an administrator is willing to hire the best secretary out of n rankable applicants

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 scheduling policy;
2: Compute t_s^{avg} and p_s for the shared scheduling 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 scheduling policy;
2: Compute t_s^{avg} and p_s for the shared scheduling 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.

for a position. The applicants are interviewed one by one in random order. A decision about each particular applicant is to be made immediately after the interview. Once rejected, an applicant cannot be recalled. During the interview, the administrator can rank the applicant among all applicants interviewed so far, but is unaware of the quality of yet unseen applicants. The question is about the optimal strategy (stopping rule) to maximize the probability of selecting the best applicant. If the decision can be deferred to the end, this can be solved by the simple maximum selection algorithm of tracking the running maximum (and who achieved it), and selecting the overall maximum at the end. The difficulty is that the decision must be made immediately.

The problem has an elegant solution. The optimal stopping rule prescribes always rejecting the first n/e applicants after the interview (where e is the base of the natural logarithm) and then stopping at the first applicant who is better than every applicant interviewed so far (or continuing to the last applicant if this never occurs). Sometimes this strategy is called the $1/e$ stopping rule, because the probability of stopping at the best applicant with this strategy is about $1/e$ already for moderate values of n. One reason why the secretary problem has received so much attention is that the optimal policy for the problem (the stopping rule) is simple and selects the single best candidate about 37% of the time, irrespective of whether there are 100 or 100 million applicants. In fact, for any value of n, the probability of selecting the best candidate when using the optimal policy is at least $1/e$.

Let W_j denote the probability of winning using an optimal rule among rules that pass up the first j applicants. Then W_jW_{j+1}, since the rule best among those that pass up the first $j+1$ applicants is available among the rules that pass up only the first j applicants. It is optimal to stop with a candidate at stage j if j/nW_j. This means that if it is optimal to stop with a candidate at j, then it is optimal to stop with a candidate at $j+1$, since $(j+1)/n > j/nW_jW_{j+1}$. Therefore, an optimal rule may be found among the rules of the following form, N_r for some $r1$: N_r : Reject the first $r1$ applicants and then accept the next relatively best applicant, if any. Such a rule is called a threshold rule with threshold r. The probability of win using N_r is

$$P_r = \sum_{k=r}^{n} \frac{1}{n} \frac{r-1}{i-1} = \frac{r-1}{n} \sum_{i=1}^{n} \frac{1}{i-1} \quad (6)$$

IV. Algorithms

A. Dominance Policy and Shared Policy

The offloading controller uses a greedy strategy to make the offloading decision. Every time an offloadable task is initiated, the offloading controller determines if it is beneficial to offload it. Because of the uncertainties inherent in the mobile environment, the offloading decision takes risk into consideration. In case a bad decision has been made, it will also adjust its strategy with new information available.

There are two main scheduling policies: (1) The dominance scheduling policy allocates a job with as many slots as possible to greedily achieve an early finishing time for each job. (2) The shared scheduling policy allocates as many jobs as possible on available machines. Here we assume a job can be executed by the way of parallel speedup. Nowadays, most cloud service providers use the FIFO scheduler. There are other advanced schedulers like priority scheduler [2] and small job first scheduler, which show good performance in minimizing the average completion time of jobs. These schedulers all use the dominance scheduling policy allocating 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, because of the sublinear speedup. For example, in Fig. 4, there are 2 jobs; assume the processing time for each task with 4 slots is 2 for each, and the the processing time for each task with 2 slots is 3 for each. Then the average completion times of different policies are equal in this example. But the shared policy has a lower machine time, and 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 aspect for the utility cost.

B. 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. If the processing speed $S(m)$ of m machines is linearly proportioned to m, then $t_i = \frac{w_i}{km_i} + t_i^{start}$. This is because t_i^{start} is the waiting time, and $\frac{w_i}{km_i}$ is the processing time of job_i. To minimize the overall time cost of all jobs,

Algorithm 3 *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 $g_{last} = N - g\lfloor N/M \rfloor$;
3: Group all the jobs in their workloads. The jobs with smaller workloads 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.

we need to consider minimizing the waiting time and the processing time of each job. We try to maximize the number of parallel machines for each job to minimize the processing time for each job. What's more, for different jobs we should apply the smallest remaining workload first policy to order the processing sequence of jobs to minimize the average waiting time. The fact is that the processing speed of a job is not always linearly related to the number of machines. Fortunately, if the processing speed is superlinearly related to the number of machines, we also want to use the maximum number of machines for each job and schedule the jobs with small workloads first. However, this kind of policy may not get the minimum utility cost when the speedup pattern is sublinear. 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,

$$\begin{aligned} t_d^{avg} &= (t_{d1} + (t_{d1} + t_{d2}) + ... + (t_{d1} + t_{d2} + ... + t_{dn}))/n \\ &= (nt_{d1} + (n-1)t_{d2} + ... + t_{dn})/n \\ &= (nw_1 + (n-1)w_2 + ... + w_n)/(nS(M)) \end{aligned} \tag{7}$$

The average finishing time for the shared scheduling policy is

$$t_s^{avg} = \sum w_i/(n \times S(\frac{M}{n})) \tag{8}$$

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

$$p_d = \frac{M \times S(1) \times \sum w_i}{S(M)} \tag{9}$$

The rent cost of machines of the shared policy is shown as,

$$p_s = \frac{M \times S(1) \times \sum w_i}{n \times S(M/n)} \tag{10}$$

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 all M machines are fully utilized. As we known $Uc_i = p_i + b \times t_i$, our objective function can be written as follows.

$$Uc_{total} = \sum_i^N Uc_i = \sum_i^N (p_i + b \times t_i) \tag{11}$$

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

$$\begin{aligned} Uc_{total} &= \sum_i^n (w_i(am_i + b)/S(m_i) + bt_i^{start}) \\ &= \sum_i^n (w_i(am_i + b)/(km_i\alpha^{m_i-1}) + bt_i^{start}) \end{aligned} \tag{12}$$

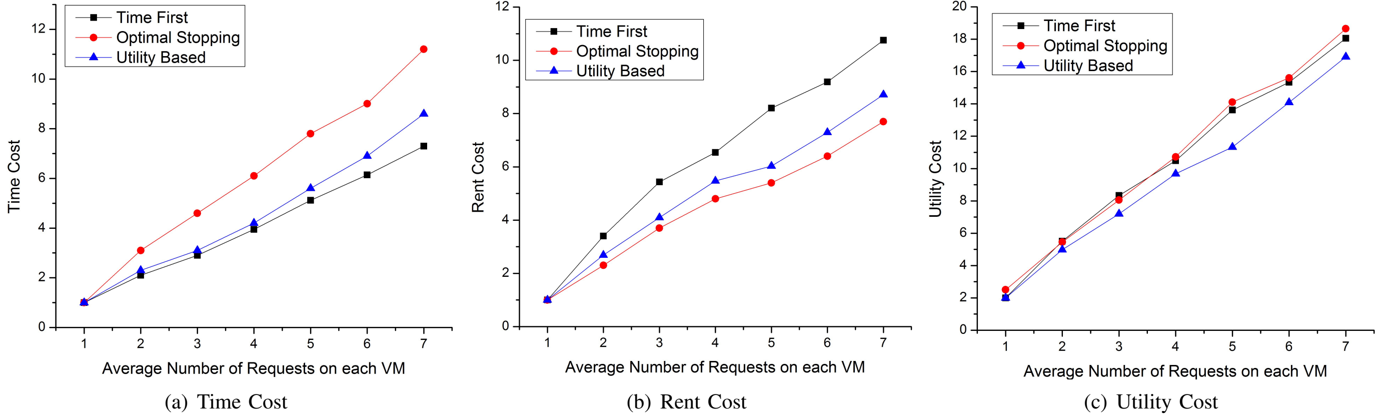
A naive way is to schedule jobs one by one. Each job is greedily allocated the optimal number of machines. The computation complexity is $O(N)$. Here, N is the total number of jobs. However, this algorithm is actually too bad to use. It is often the case that one job will use up all the machines at a 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 a better 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 $\sum_j^J g_j = N$.

Here we provide three group utility algorithms. For the first one, 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. 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. The worst time complexity of this algorithm is $O(M^N)$.

V. EXPERIMENTS

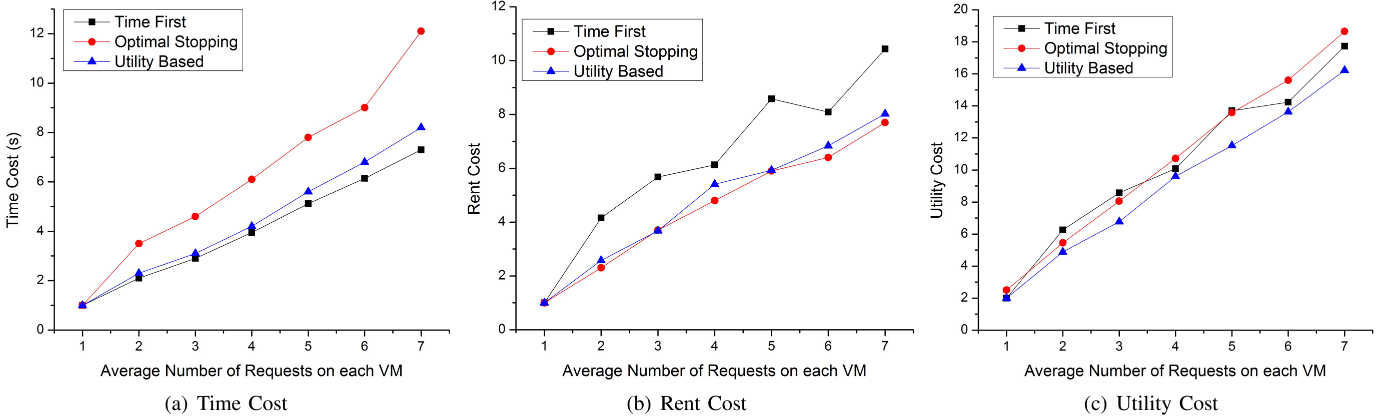
One thing that stands out is that, in general, the more runs of a job that occur, the faster that job runs. While a job does not always get better with every run, the general trend is for jobs to run faster after a few runs. We believe that there are some likely reasons for this to occur. The first reason that this may happen is due to the fact that the jobs may already have their data loaded into memory. Since the jobs have just recently been run within the cluster, their data still may be present in RAM. This could mean we could see a substantial increase in speed.

Another reason this might occur is because the Hadoop scheduler might be getting used to the job and the data. It is



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

Fig. 5. Simulation results of 3 algorithms under hybrid speedup pattern.



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

Fig. 6. Trace-based results of 3 algorithms under hybrid speedup pattern.

an adaptive and flexible scheduler. Due to this, the more that we run the job with the same data, the more it may adapt to both this data and this job. Unfortunately, determining the exact cause of this will take more time than what is available for this paper. Therefore all we are able to do is give our reasoning as to why this might have occurred. The second thing that we noticed is that the consistency from run to run seemed to increase with the number of nodes. Next, we evaluate our algorithm using real-world access traces [3].

A. Evaluation Results

It is clear that the price first policy has 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 use a few machines with minimal cost for every job, so there are many machines unassigned when the number of jobs are very small. However, when combining both time and rent cost, neither the price first policy nor the time first policy are the best. What's more, we find that optimal stopping policy share similar property as price first policy. Thus, we only provide time first policy in Fig. 5 and Fig. 6.

In Fig. 5(a) and Fig. 6(a), the average finishing time increase with the increasing number of requests on each machine. Obviously, more machines will decrease the average finishing time of a job. But there is a another trend that the total rent cost increase with more requests added on each machine as shown in Fig. 5(b) and Fig. 6(b). Obviously, adding more machines cannot help decreasing the rent cost. Here we need to balance the rent cost increase and time cost increase. The overall utility cost is shown in Fig. 5(c) and Fig. 6(c). Here we use the Word Count application for all experiments. The data set for Fig. 6 consists of 8 access traces, each of which is composed of access requests in 2 days. We use these time stamps of access requests as the start time of mobile applications on various mobile devices. We evaluate the performance through simulation. The average number of requests from the same user is very low in the traces, indicating extremely high cost of CloneCloud.

subsection Simulation Summary From our experiment, the proper scheduling policy might avoid using only one machine, which provides a high cost of time and price. It might also try to assign each job with the number of machines in the good speed range (linear or super-linear range). Once the range is settled, the utility-based policy can be applied. The utility-based policy is now a batch launch policy; the number of jobs started in a batch is decided by the proper speed range of parallel machines and the total number of machines. Then the utility-based shifting scheduling policy will decide which kind of scheduling method is good for those jobs in a batch. Then we test our algorithms by using the results we get from the real traces. Three algorithms are shown in Fig. 6. As a result, the utility-based policy is powerful in maximizing the utility

gains of the cloud clusters.

VI. Related Work

A fundamental aspect for cloud providers is reducing data center costs while guaranteeing the promised Service Level Agreement (SLA) [1] to cloud consumers. Several recent works [4][5] addressed the VM assignment problem by minimizing the average finishing time of jobs assigned to machines, and several algorithms [6][7] have been proposed with the objective of maximizing the utilization of the virtual machines. Closer to our work, MAUI [8] enables mobile applications to reduce the energy consumption through automated offloading. Similarly, CloneCloud [9] can minimize either energy consumption or execution time of mobile applications by automatically identifying compute-intensive parts of those applications. ThinkAir [10] enables the offloading of parallel tasks with server-side support. These systems focus on how to enable computation offloading for mobile devices. The challenges of computation offloading with variable connectivity have been identified in [11]. A system, Serendipity [29], was designed for computation offloading among intermittently connected mobile devices. In contrast, COSMOS proposes techniques to handle the variable connectivity for offloading to a cloud.

The concept of cyber foraging [12], i.e., dynamically augmenting mobile devices with resource-rich infrastructure, was proposed more than a decade ago. Since then, significant work has been done to augment the capacity of resource-constrained mobile devices using computation offloading [13]. A related technique proposes the use of cloudlets which provide software instantiated in real-time on nearby computational resources [14]. Our work is also related to studies on cloud resource management. This problem is intensively studied in the context of power saving in data centers [15] [16]. For example, Lu et al. [15] uses reactive approaches to manage the number of active servers based on current request rate. Gandhi et al. [16] investigate policies for dynamic resource management when the servers have large setup time. COSMOS is different from these in three major aspects. First, they minimize the cost of power consumption, whereas COSMOS reduces the cost of leasing cloud resources. Second, in COSMOS computation tasks may be offloaded to the cloud or be executed on local devices, while in data centers services are always provided by servers. Third, COSMOS also needs to handle variable network connectivity of mobile devices, unnecessary for data center.

VII. Conclusion and Future Work

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. All three algorithms are for batched start group jobs, but each method has 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 remaining with respect to the design of algorithms that minimize response time.

VIII. Acknowledgement

This work is supported in part by NSF grants CNS 149860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, ECCS 1231461, ECCS 1128209, and CNS 1138963.

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] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Proceedings of JSSPP 2010*, pp. 110–131.

[3] ftp://ftp.ircache.net/Traces/.

[4] 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.

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

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

[7] ——, "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.

[8] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of ACM MobiSys 2010*, pp. 49–62.

[9] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of EuroSys 2011*, pp. 301–314.

[10] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of IEEE INFOCOM 2012*, pp. 945–953.

[11] J. Flinn, "Cyber foraging: Bridging mobile and cloud computing," *Synthesis Lectures on Mobile and Pervasive Computing*, vol. 7, no. 2, pp. 1–103, 2012.

[12] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, 2001.

[13] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *Proceedings of ACM MobiSys 2003*, pp. 273–286.

[14] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[15] Y.-H. Lu, E.-Y. Chung, T. Simunic, L. Benini, and G. De Micheli, "Quantitative comparison of power management algorithms," in *Proceedings of DATE 2000*, pp. 20–26.

[16] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems*, vol. 30, no. 4, p. 14, 2012.