Resource Provision for Batch and Interactive Workloads in Data Centers

Ting-Wei Chang*, Ching-Chi Lin*§, Pangfeng Liu*†, Jan-Jan Wu§, Chia-Chun Shih‡, Chao-Wen Huang‡
* Department of Computer Science and Information Engineering, National Taiwan University
{r02922100, d00922019, pangfeng}@csie.ntu.edu.tw
† The Graduate Institute of Networking and Multimedia, National Taiwan University
{pangfeng}@csie.ntu.edu.tw
‡ Chunghwa Telecom Laboratories
{ccshih, huangcw}@cht.com.tw
§ Institute of Information Science, Academia Sinica
Research Center for Information Technology Innovation, Academia Sinica
{deathsimon, wuj}@iis.sinica.edu.tw

***Abstract*—In this paper we describe a scheduling framework that allocates resources to both batch jobs and interactive jobs simultaneously in a private cloud with a static amount of resources. In the system, every job has an individual service level agreement (SLA), and violating the SLA incurs penalty. We propose a model to formally quantify the SLA violation penalty of both batch and interactive jobs. The analysis on the interactive jobs focuses on queuing analysis and response time. The analysis on batch jobs focuses on the non-preemptive job scheduling for multiple processing units. Based on this model we also propose algorithms to estimate the penalty for both batch jobs and interactive jobs, and algorithms that reduce the total SLA violation penalty. Our experiment results suggest that our system effectively reduces the total penalty by allocating the right amount of resources to heterogeneous jobs in a private cloud system.**

Keywords*—*Scheduling algorithms; Client-sever systems; Parallel processing; Multiprocessing systems

I. Introduction

Cloud computing has become an important topic in the past few years. Many enterprises have built their own private clouds for production and providing services because private clouds are easier to utilize and more secure. [1], [2]

In private clouds, the total amount of resources is fixed most of the time. Private cloud owners purchase new hardware according to their schedule. They will buy machines for every given fixed period of time, instead of purchasing machines in a dynamic and on-demand manner. In contrast consumer can request more resources on demand in public clouds. In addition, most public cloud systems provide *auto-scaling* that will automatically increase, or decrease the number of machines to adapt to the dynamic requirement of applications. However, in private clouds the total amount of resources is limited by the size of the data center. Consequently it is critical to schedule the *limited* computing resources to applications in order to satisfy various service requirements.

The applications in a cloud may have varying characteristics and requirements. We categorize the applications into two groups – *batch jobs* and *interactive jobs*. Batch jobs, e.g., file processing jobs, have a *soft deadline*. That is, they are required to finish within a given time period, but the requirement is not absolute. There is usually a penalty associated with the deadline. Therefore if there are other high priority jobs suddenly admitted into the system, the system may decide to pay the penalty and let the high priority jobs run first. In summary, the goal of scheduling batch jobs is to have high *throughput*, instead of getting fast responses.

The interactive jobs, e.g., web servers, have constraints on their *response time*, i.e., the time between job submission and completion. It is crucial for these web application, e.g., gmail or Google document, to respond to user input in a timely manner, otherwise user experience will be unacceptable. That is, when the requests come to these interactive applications, they must be processed and responded within a reasonable *respond time*. In summary, the goal of scheduling interactive jobs is to have low *latency*.

Both interactive and batch jobs have quality of service requirements. We must ensure that batch jobs will finish before their deadlines, and interactive jobs can respond to requests of services as fast as possible. If these constraints are not met, the data center will suffer penalty. The penalty is in the form of service level requirements (SLA). Violation of SLA results in penalty, and the goal of a data center management system is to minimize this penalty while scheduling both interactive and batch jobs.

In this paper, we propose a cloud resource management framework to dynamically adjust the resource allocation and to dispatch workload to machines. We prove that the problem is NP-hard and propose a heuristic algorithm to allocate resources to jobs. The algorithm will consider the status of machines, the status of jobs, and the constraints of jobs. The key idea is to find the best schedules for batch jobs and interactive jobs *separately*, and then merge the schedules to obtain the final resource allocation plan.

The contributions of this paper are summarized as follow.

- A framework designed for private clouds, which that have static amount of resources, to allocate resources to batch and interactive jobs.

- A theoretical analysis on the expected penalty on both interactive and batch jobs. The analysis on the interactive jobs focuses on queuing analysis and response time. The analysis on batch jobs focuses on the non-preemptive job scheduling for multiple processing units.
- A heuristic algorithm that considers the constraints of both batch and interactive jobs, and minimizes the total penalties.

The rest of the paper is organized as follows. Section III describes the system architecture and Section IV describes the model of jobs. Section V describes the resource provisioning problem and our algorithm. Section VI describes experiment results and Section VII concludes.

II. Related Work

There are several systems in the literature that schedule batch jobs and interactive jobs for clusters. Li et al. [3] proposed a system to deploy both interactive services and batch jobs onto public clouds while trying to minimize rental fees. The dependency of tasks in a batch job is represented by a directed acyclic graph (DAG). In order to speed up batch jobs, they assign individual deadline to tasks according to the DAG. They claimed that their algorithm reduced the rental fees by 11% compared to the second best algorithm.

Google's Borg [4] is a large scale cluster management system that allocates resources based on priorities of jobs. Borg scans through pending tasks from high to low priority, and within the same priority jobs are run in a round-robin manner. Borg partitions priorities into several non-overlapping bands and prevents higher-priority jobs from preempting lower-priority jobs in the same priority band. By mixing user-facing workloads and batch workloads in the same cluster, Borg can reduce the number of machines required. If user-facing workloads and batch workloads are segregated, $20-30\%$ more machines is needed to run the same workload [4].

Garg et al. [5] proposed a scheduling system that handles both batch jobs and interactive jobs. They predict resource demands of jobs with Artificial Neural Network [6] (ANN), and assign batch job virtual machines and interactive job virtual machines to the same physical machine to increase resource utilization.

Our work focused on reducing the penalty of SLA violation while scheduling both batch jobs and interactive jobs simultaneously. Li et al. [3] focused on reducing the rental fee of scheduling jobs onto public clouds. Borg [4] used priority-based scheduling and focused on achieving high utilization. Garg et al. [5] also focused on resource utilization.

In addition to scheduling both batch and interactive jobs, there are works that schedule only batch workloads with soft deadline. Yeo and Buyya presented LibraSLA [7], a proportional share allocation technique. LibraSLA implements admission control to ensure that more utility is achieved if a new job is accepted.

Wu, Garg, and Buyya [8] proposed resource allocation algorithms for SaaS providers who want to minimize infrastructure cost and SLA violations.

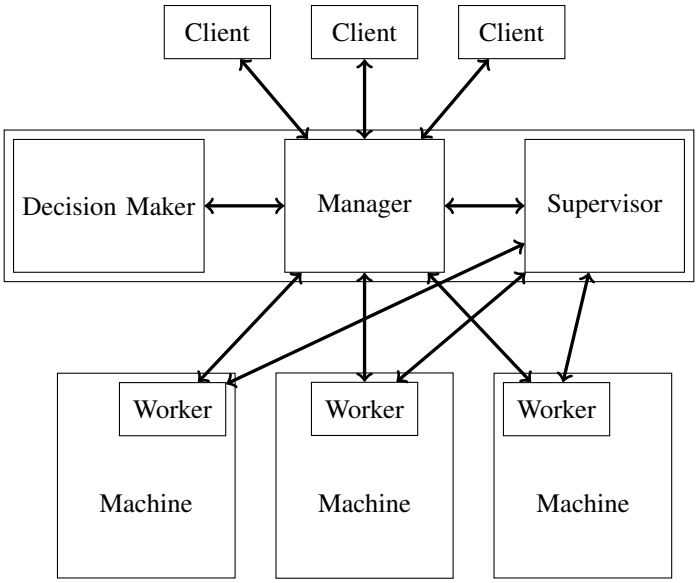

Fig. 1: System Architecture

LibraSLA [7] implements admission control while our work handle overload situations by reducing penalties. The algorithm presented by Wu, Garg, and Buyya [8] is designed for SaaS providers who can rent more resource from IaaS providers while our work focus on private clouds with fixed amount of resources.

There are some works [9], [10], [11] in which every job has a time utility function that computes the utility gained as a function of completion time. The purpose of these works is to maximize the accrued utility. However, those systems are allowed to abort jobs that have low expected utility while our model is designed to complete all jobs eventually.

III. Architecture

Our system consists of five parts: a *supervisor*, a *decision maker*, a *manager*, multiple *workers*, and multiple *clients*. Figure 1 illustrates the architecture of the proposed framework.

A. Supervisor

The supervisor collects status of all workers and gathers performance profiles of jobs from workers. The status of a worker includes the tasks the worker is executing, the total number of processing units of the worker and the number of available processing units of the worker. A performance profile of a batch job consists of the average execution time of a task and other information. A performance profile of an interactive job consists of the average processing time of a request, the number of queued requests, and the input rate of incoming requests.

B. Decision Maker

The decision maker is the core of our system because the it decides how resources are allocated to jobs. When requested by the manager, the decision maker generates an execution plan according to arguments given by the manager, e.g., the status

of workers, the current resource allocation, the performance profiles of jobs, and etc.

An execution plan consists of multiple sub-plans each for a worker. A sub-plan for a worker indicates which running tasks should be killed, and which jobs should be executed. After the execution plan is sent to the manager, the manager distributes the sub-plans to workers and dispatches tasks to workers according to the sub-plan.

C. Manager

The manager receives jobs from the clients, manages information about jobs, launches rescheduling procedure, and dispatches tasks to workers. Whenever a client submits a job or a task completes, the manager starts a reschedule procedure in the following sequence.

1) The manager asks the supervisor to provide performance profiles of jobs and status of workers. After the supervisor retrieves these information from workers, the supervisor relays them to the manager.
2) The manager sends information about jobs, performance profiles of jobs, and status of all workers to the decision maker.
3) The decision maker generates an execution plan based on the given information.
4) The manager distributes the execution plan to workers and dispatches tasks to workers based on the execution plan.

D. Worker

Every physical machine has a worker process that manages the computing resources on the machine. Workers report their status and performance profiles of jobs to the supervisor and execute tasks assigned by the manager. A worker can execute multiple tasks simultaneously and spawn threads to manage the running tasks.

Workers collect performance data to generate performance profile for both batch jobs and interactive jobs. When a batch job completes, the worker generates a profile of the batch job. While an interactive job is executing, the worker keeps collecting performance data and generating new performance profiles.

E. Client

A client provides a programming interface for users to submit jobs to our system and synchronizes submitted jobs. User can interact with the manager via a client instance created from the library. When submitting a job, the user must provide information about computing the cost of the job and an initial profile of the job. The client can synchronize submitted jobs therefore enforcing dependency among jobs if needed.

IV. Models

Our system can handle two kinds of jobs: *batch jobs* and *interactive jobs*. A batch job consists of many tasks, and each task processes an input file. An interactive job processes continuously incoming requests.

The goal of job scheduling is to minimize the total *penalty*. Both batch jobs and interactive jobs may incur penalty. The total penalty is the sum of the penalty from batch jobs and interactive jobs.

A. Batch Job

A batch job consists of many *tasks* of roughly the same execution time. A task is a command that executes a program and specifies an input file to the program. Since we can control the length of input files to be roughly the same, the tasks for processing these files will have roughly the same execution time. The program terminates after it finishes processing the input file and writing outputs to an output file. A user must define all tasks of a batch job before he submits the job. Tasks of the same batch job are *independent*, so they can be executed in parallel. When all tasks of a batch job complete, the batch job completes.

The system runs batch jobs as follows. The system will allocate m_b processing units for batch jobs. A Task can run on *any* processing unit in *any* order because it is independent from all other tasks, but two tasks *cannot* run on the same processing unit simultaneously, and a task must run to its completion without interruption. The *completion* time of a job is the completion time of the task that completes *last*.

A batch job may incur *penalty* according to its *completion time* and its *soft deadline*. Every batch job has a completion time c, a soft deadline d and a penalty rate p (penalty per time unit). If the batch job completes before its deadline, the delay is 0; otherwise, the delay is the amount of time from the completion time c to the soft deadline d, i.e., $c-d$, where $c > d$. The penalty of a batch job, denoted by P, is the product of the penalty rate p and the delay. Some research in the literature uses linear penalty model as well [7], [12].

$$P = p \times \max(0, c - d) \tag{1}$$

The goal of scheduling batch jobs is to arrange tasks of all batch jobs so that the total penalty is minimized. Here we assume that there are m_b processing units, and the scheduling determines when and where all tasks will run in all m_b processing units, under the constraint that no two tasks run simultaneously on a processing unit, and a task must run to its end without interruption.

B. Interactive Job

An interactive job is a program that runs forever serving requests. The program continuously receives requests from users, processes the requests, and responds to users. The program of an interactive job is state-less, so we can run multiple instances of the program at the same time on multiple machines to reduce average response time. The requests are pushed into a queue, and the programs pop requests from the queue.

The penalty of an interactive job is based on its *response time* within a time window. Let p be the penalty rate of an interactive job, r be the fraction threshold, and ϵ be the response time threshold. If the response time of a request is less than the threshold ϵ, the response is *satisfying*; otherwise

the response is *non-satisfying*. Let $f(m)$ be the *fraction* of satisfying responses within a time window when given the number of processing units m. It is easy to see that f should be an non-decreasing function of m, i.e., when an interactive job is given more processing units, it should provide more satisfying responses. If the the fraction of satisfying responses f is greater than the fraction threshold r, the penalty is 0; otherwise, the penalty is the product of the penalty rate p, and the difference between the fraction threshold r and the fraction of satisfying response f.

$$P = p \times \max(0, r - f(m)) \tag{2}$$

The goal of scheduling interactive jobs is to arrange tasks of all interactive jobs so that the total penalty is minimized. Here we assume that there are m_i processing units for interactive jobs, and the scheduling determines the number of processing units allocated to each interactive jobs, so that the total penalty is minimized. Note that all interactive jobs will use some processing units, and each of them will give $f(m)$ fraction of satisfying responses if it is given m processing units.

To summarize, our scheduling algorithm must determine the following to minimize the total penalty from both interactive and batch jobs.

- The number of processing units allocated to batch jobs m_b and interactive jobs m_i, under the constraint that the sum of m_b and m_i is no more than the total number of processing units.
- When and where tasks from all batch jobs should start, given the penalty rate, the deadline, and the execution time of all tasks.
- The number of processing units allocated to each interactive job, given the $f(m)$ function that determines the satisfying fraction of requests, when given m processing units.

V. Resource Provisioning

A. Problem Definition

We now formally define the problem. We are given a set of batch jobs B, a set of interactive jobs I, the size of a time window w, the number of processing units m, and a penalty C. A job $j \in B$ has $n(j)$ tasks, each has an average execution time $\mu(j)$, a penalty rate $p(j)$, and a soft deadline $d(j)$. A job $j \in I$ has a penalty rate $p(j)$, a response time threshold $\epsilon(j)$, a fraction threshold $r(j)$, a satisfying fraction function $f(j, m)$. Is there a schedule to run all jobs with the total penalty incurred before all batch jobs complete no more than C? We will refer to this scheduling problem as *Batch and Interactive Job Scheduling*.

Theorem 1. *The batch and interactive job scheduling is NP-complete.*

Proof:

It is easy to see that the batch and interactive job problem is in NP because a non-deterministic Turing machine can always guess a scheduling and verify its feasibility in polynomial time.

We reduce the *sequencing to minimize weighted tardiness problem* [13] to our batch and interactive job scheduling problem to prove that our problem is NP-hard. The minimize weighted tardiness problem is as follows. Given a set T of tasks, for each task $t \in T$, a length $l(t)$, a weight $w(t)$, and a deadline $d(t)$, and a positive integer K. Is there a one-processor schedule σ for T such that the sum, taken over all $t \in T$ satisfying $\sigma(t)+l(t) > d(t)$, of $(\sigma(t)+l(t)-d(t)) \times w(t)$ is K or less? In other words, the problems demands a schedule that the sum of penalty is bounded by K. It is easy to see that sequencing to minimize weighted tardiness problem is a special case of our batch and interactive job scheduling, where all jobs are batch jobs. The theorem follows.

■

B. Algorithm

We design an algorithm for the batch and interactive job scheduling problem. Note that from Theorem 1 the problem cannot be solved optimally, so we will use heuristics to obtain schedule with reasonable quality.

Our algorithm has three steps. The first step is to compute the minimum penalty of interactive jobs in the next window under different number of allocated processing units. This step corresponds to the third output of the the scheduling result summary at the end of the previous section. The second step is to compute the minimum penalty of batch jobs per time window under different number of allocated processing units. This step corresponds to the second output of the the scheduling result summary. Finally after knowing the penalty of interactive and batch jobs under different number of processing units (m_i and m_b), the third step finds the numbers of processing units that should be allocated to interactive and batch jobs, so that the total penalty is minimum.

1) Estimating the Cost of Interactive Jobs: We now explain how to compute the number of processing units that should allocated to interactive jobs. We first compute the $f(j, m)$ function that determines the satisfying fraction of requests for every interactive job j and every number of processing units m. Then we use a dynamic programming to compute the function determines the minimum total penalty from all interactive jobs for every number of processing units.

We now estimate $f(j, m)$, the satisfying fraction function of job j when given m processing units. Let $q_0(j)$ be the number of requests in the queue initially, $i(j)$ be the incoming rate of requests for job j, and $\mu(j)$ be the average request processing time of job j. Then the number of queued requests a request arrived at t will see in front of it, denoted as $q(j, t, m)$, is the number of requests previously in the queue, plus those that came before t, minus those that have finished before t.

$$q(j, t, m) = \max(0, q_0(j) + i(j) \times t - \frac{m}{\mu(j)} \times t) \tag{3}$$

We use $w(j, t, m)$ to denote the *waiting time* of a request of job j that joined the queue at time t. $w(j, t, m)$ is estimates as in Equation 4 because job j has to wait until all previously

queued requests complete.

$$w(j,t,m) = q(j,t,m)\frac{\mu(j)}{m} \tag{4}$$

We now consider two cases. In the first case the incoming rate of requests is greater than or equal to the consuming rate of requests, i.e., $i(j) \geq \frac{m}{\mu(j)}$. In this case the waiting time $w(j,t,m)$ will be a non-decreasing function of t. That is, later jobs will wait longer because jobs will pile up in the queue. In the second case the incoming rate of requests is less than the consuming rate of requests, and $w(j,t,m)$ will be a increasing function of t. That is, the waiting time for later jobs will be shorter because the queue is getting shorter.

We now estimate the number of requests that are satisfying, so as to compute the fraction of satisfying requests. Since a request is satisfying if the sum of its waiting time and processing time is less than its response time threshold ϵ, That is, as in Inequality 5.

$$w(j,t,m) + \mu(j) \leq \epsilon(j) \tag{5}$$

It is easy to see from Inequality 5 that the arrival time t determines whether a task is satisfying or not. To clarify this relation we set Inequality 5 to be equal, and solve for $t = t^*$. It is easy to see that depending on whether $w(j,t,m)$ is an increasing or decreasing function of t, and whether t is before or after t^*, a task can be either satisfying or non-satisfying. We analyze this in two cases – t^* is in $[0,W]$ or t^* is not in $[0,W]$.

a) t^* is in $[0,W]$: It is easy to see that when the incoming rate of requests is greater than or equal to the consuming rate of requests, any job came after t^* will be non-satisfying because jobs are piling up in the queue. On the other hand, when the incoming rate of requests is less than or equal to the consuming rate of requests, any job came after t^* will be *satisfying* because the length of the queue has reduced to the point that the response time is short enough. Since we assume that t is uniformly distributed between 0 and W, the $f(j,m)$ is computes as follows.

$$f(j,m) = \begin{cases} \frac{t^*}{W} & \text{if } i(j) \geq \frac{m}{\mu(j)} \\ \frac{W - t^*}{W} & \text{otherwise} \end{cases}$$

b) t^* is not in $[0,W]$: If t^* is not in $[0,W]$, then the queue is either too short so that every request arriving between 0 and W is satisfying, or too long so that every request arriving between 0 and W is non-satisfying. In these extreme cases the $f(j,m)$ is either 1 or 0.

Recall that if the the fraction of satisfying response f is greater than the fraction threshold r, the penalty is 0; otherwise, the penalty is the product of the penalty rate p and the difference between the fraction threshold r and the fraction of satisfying response f. Consequently the expected penalty of running job j with m processing units in the next time window, denoted as $c(j,m)$, is defined as in Equation 6.

$$c(j,m) = p(j)\max(0, r(j) - f(j,m)) \tag{6}$$

After knowing the penalty function $c(j,m)$ of running job j with m processing units, we need to determine the best way to distribute a given number (m_i) of processing units to all interactive jobs so that the total penalty is minimized. We will use a dynamic programming to do this.

We define $D(i,m)$ to be the minimum penalty of running the *first* i jobs with m processing units. Before we get to the final solution we first describe the base case of the dynamic programming, i.e., we consider $D(1,m)$, where only the first job is considered. Let the number of jobs be k, then $c(j_i,m), 1 \leq i \leq k$, is the expected cost of the interactive job j_i running on m processing units, where $0 \leq m \leq m_i$. It is easy to see that $D(1,m)$ is by definition, $c(j_1,m)$.

$$D(1,m) = c(j_1,m) \tag{7}$$

Now we describe the recursive relation of D. For $i > 1$, we can allocate h processing units to the first $i-1$ jobs, and $m-h$ processing units to the i-th job. We try all possible h's and choose the one that has the minimum penalty, as in Equation 8.

$$D(i,m) = \min_{0 \leq h \leq m} (D(i-1, m-h) + c(j_i,h)) \tag{8}$$

It is easy to see that the solution will be $D(k,m_i)$.

2) Estimating the Penalty of Batch Jobs: We now estimate the penalty of all batch jobs under different number of processing units. For each number of processing units m, it takes three steps to estimate the total penalty and schedule jobs among processing units. First, we decide an order by which batch jobs will be assigned to processing units. Second, we schedule jobs to run on the processing units by assigning the starting time and processing unit for each job. Finally we estimate the total penalty of the schedule.

We determine the order of job assignment by a greedy method.

a) The Greedy Method: Recall that if a batch job j starts execution at time t with m processing units, the *delay* is defined as follows.

$$L(j,t,m) = \max(0, t + \mu(j)\lceil\frac{n(j)}{m}\rceil - d(j)) \tag{9}$$

Consequently the penalty, denoted as $c(j,t,m)$, is as follows.

$$c(j,t,m) = p(j)L(j,t,m) \tag{10}$$

Now if this job j is further delayed by δt of time, the penalty will *increase* by $p(j)(L(j,t+\delta t,m)-L(j,t,m))$. We use δc to denote this increase.

$$\delta c(j,t,\delta t,m) = p(j)(L(j,t+\delta t,m) - L(j,t,m)) \quad (11)$$

Now if we select a batch job j' to run next, it will run for $\mu(j')\lceil\frac{n(j')}{m}\rceil$ time. In other words, j' could potentially delay all other jobs and increase their penalties. Equation 12 sums up the increased penalty of all other jobs in B if we select j' to run next.

$$\sum_{j\in B-j'} \delta c(j,t,\mu(j')\lceil\frac{n(j')}{m}\rceil,m) \quad (12)$$

The greedy algorithm works as follows. Initially B has all the jobs. We then select a job j' that *minimizes* Equation 12, remove j' from B, and increase t by $\mu(j')\lceil\frac{n(j')}{m}\rceil$, then repeat the process. The order we assign jobs will be the order we select jobs from this greedy method.

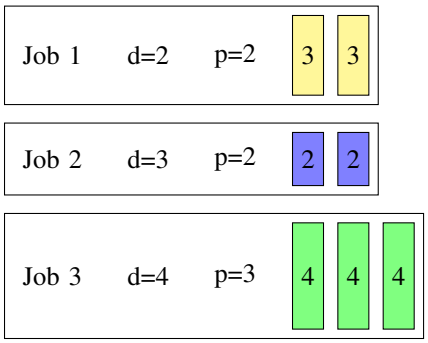

Fig. 2: A set of batch jobs

We illustrate how the greedy algorithm determines job order with an example. We are given three processing units and a set of batch jobs B as in Figure 2. There are three jobs. Job one has 2 tasks of length 3, job 2 has 2 tasks of length 2, and job 3 has three jobs of length 4. The deadlines of jobs are 2, 3, and 4 respectively. The number of processing units is 3.

The greedy method determine the task order as (j_2, j_3, j_1) for tasks in Figure 2. When the greedy method selects the first job, it computes the increased penalty (Equation 12) to others, due to j_1, j_2, and j_3, are 13, 10, and 14, respectively. Consequently j_2 is selected as the first job because it causes the minimum penalty to others. When the greedy heuristic selects the second job, the increased penalty due to j_1 and j_3 are 9 and 8 respectively, so j_3 is selected next. The job j_1 is selected last.

We can also use other strategies to order jobs. Candidates include the Earliest Deadline First strategy (EDF) [14], the Least Slack Time First strategy (LST) [15], the Least Slack Time Rate First strategy (LSTR) [16], and the Highest Penalty Rate First strategy (HPRF) algorithms. The EDF strategy sorts jobs by their deadlines in ascending order. The LST strategy sorts the jobs by their slack time, which is $d(j)-\mu(j)\lceil\frac{n(j)}{m}\rceil$, in ascending order. The LSTR strategy sorts the jobs by the ratio between their slack time and their deadlines in ascending order. The HPRF strategy sorts the jobs by their penalty rate in descending order. We will compare our greedy method with these strategies with experiments.

b) Schedule Jobs: After we determine the order to schedule jobs, we will schedule jobs to processing units by determining the starting time and processing units to all tasks. We will be given a list of jobs sorted as described earlier. Initially all processing units are available. Whenever a processing unit p becomes available, we pick a task t from the first job in the list that still has tasks, schedule t to run on p, and update the available time of p to be μ later in the future. Recall that μ is the average execution time of the task. The scheduling ends when all tasks from all jobs are assigned to processing units.

We illustrate the scheduling process, which is based on an job order, with an example. We consider the batch jobs in Figure 2, and the order of jobs from the greedy method is (j_2, j_3, j_1). Because all processing units are available at time 0, we schedule one tasks of j_2 to processing unit 1, the other tasks of j_2 to processing unit 2, both run in the time interval $(0, 2]$. The we schedule a task of j_3 to processing unit 3 in the time interval $(0, 4]$. When processing unit 1 and 2 become available at time 2, we schedule the other two tasks of j_3 to them, and run the tasks in the time interval $(2, 6]$. Finally, we schedule the tasks of j_1 to processing unit 3 and 1, and run the tasks in the time interval $(4, 7]$ and $(6, 9]$ respectively. The schedule is shown in Figure 3.

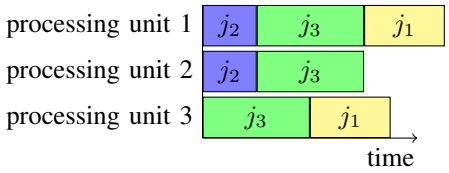

Fig. 3: The schedule from the greedy heuristic

After scheduling all tasks to processing units we compute the total penalty of batch jobs. Since we have assigned all tasks to processing units, we can calculate the completion time of all jobs. Then from Equation 10 we compute the penalty of each job and the total penalty is simply the sum of penalty of all jobs.

Before we compare the penalty of batch and interactive jobs, we need to put them in the same scale. For interactive jobs the penalty is calculated within a time window of length W, but the penalty of batch jobs is calculated from the beginning to the end of its execution. Therefore we scale the penalty of batch jobs by the ratio between the size of the time window W and its completion time.

3) Determine m_i and m_b: The final step is to find the m_i and m_b that minimize the total penalty. That is, we decide the number of processing units that will be allocated to interactive and batch jobs respectively.

From the first step, we compute a function $c_I(m)$ that estimates the minimum penalty of all interactive jobs with m

processing units. From the second step, we compute a function $c_B(m)$ that estimates the minimum penalty of batch jobs with m processing units. Now given the total number of processing units m, the minimal penalty P can be obtained by trying all possible numbers of processing units that will be allocated to interactive jobs, and all the rest to batch jobs, as in Equation 13.

$$P = \min_{0 \le i \le m} (c_I(i) + c_B(m - i)) \tag{13}$$

VI. Experiment

A. Hardware Settings

Our experimental environment consists of one manager node, one decision maker node, one supervisor node, one client node, and four worker nodes. Each node has two quad-core Intel Xeon CPU X5570 running at 2.93GHz, and has 768MB of memory. Each worker node has 16 processing units, i.e., 64 processing units in total.

B. Workload Traces

We use SDSC-Par96 trace log [17] for our batch job experiments. This log has a year's worth of accounting records for the 416-node Intel Paragon located at the San Diego Supercomputer Center (SDSC), and contains 32135 jobs in standard workload format [18]. The execution time of jobs from SDSC-Par96 trace varies dramatically. Some jobs only last for a few seconds, while some other jobs last for more than three days.

Due to the large volume of the SDSC log we sample jobs from it. Each sample contains 313 consecutive jobs. For simulation efficiency, we scaled down the execution time and the time between jobs by a factor of 0.001. In our simulation, the number of tasks of a job is set to the number of processors allocated to the corresponding job in SDSC log.

We set the parameters of a task as follows. The execution time of a task in our experiment is set to the actual execution time of the task in the log. The deadline of a task in our experiment is set to the its actual completion time in the log, i.e., the deadline is set so that if our system does *not* make this job wait longer in our system than in the trace, it will not have penalty. Finally the penalty rate of a job is randomly generated.

We use Calgary-HTTP and Saskatchewan-HTTP trace log from The Internet Traffic Archive [19] for our interactive job experiments. The logs contain a year's worth and seven month's worth of records of HTTP requests respectively. The Calgary-HTTP log contains 724693 records and the Saskatchewan-HTTP log contains 2407086 records. A record consists of the timestamp of the request, the HTTP status code, and the size of the response. We set the processing time of a request as the time needed to read such amount of data from a hard drive with data transfer rate of 1030 Mbits/sec and a rotational latency of 8 milliseconds.

Due to the large time scale of the HTTP logs we sample records from it. Each sample contains 100 hour's worth of records. Since the requests are quite sparse, we scaled down the time between requests by a factor of 0.01.

C. Experiment Results

We conducted three sets of experiments. The first experiment evaluates the accuracy of our batch job on the penalty estimation. The second experiment compares our greedy algorithm against other methods for scheduling batch jobs, based on the average penalty. The third experiment compares the SLA violation penalty under mixed interactive and batch jobs.

1) Penalty Estimation of Batch Jobs: In the first experiment we evaluate the penalty estimation accuracy on batch job. Since penalty estimation will be affected by later submitted jobs, for the purpose of evaluation, we assume that all jobs are available at the beginning. We also randomly set the penalty rates of task between 1 and 1000.

We use the greedy algorithm to schedule a sample of batch jobs from the trace, and record the penalty estimation as time progresses. Note that the penalty estimation is a continuous process – whenever a task finishes, the system will re-schedule jobs, and this rescheduling will need to estimate the penalties of all incomplete jobs. During the rescheduling, the decision maker estimates the expected penalty of every job, and records its the identity, estimated penalty, and the timestamp. In addition, the manager records the final actual penalty of a job when it finishes. Consequently we have the estimation of penalty as time progresses, and the final penalty of a job.

We compute the ratio between the estimated penalty and final penalty at different time for jobs with non-zero final penalty as time progresses. Figure 4 shows the ratio of eight selected jobs as time progresses. In addition, we also compute the ratio between the total estimated penalty and total final penalty of incomplete batch jobs as time progresses, and show the ratio in Figure 5.

From Figure 4 and 5 we observe that our penalty estimation quickly converges to the correct value as the execution proceeds. The entire execution takes about 1500 seconds to complete. At the beginning of the execution, the overall accuracy was about 0.73. After 46.6 seconds, about 3% of the entire execution, there were 160 incomplete jobs left, and the overall accuracy reached 0.80. After 91.3 seconds, about 6% of the entire execution, there were only 100 incomplete batch jobs left, and the overall accuracy reached 0.9. After 294.6 seconds, about 20% of the entire execution, there were 48 incomplete batch jobs left, and the overall accuracy reached 0.95.

We note that the number of jobs drop dramatically in the beginning of the execution. In fact the system executed almost half of the jobs in only 3% of the time. There are two reasons for this fast reduction. First the execution time of jobs from SDSC-Par96 trace varies dramatically. Some jobs only last for a few seconds, while some other jobs last for more than three days. Second, our greedy algorithm will evaluate a job based on the penalty it causes other jobs (Equation 12), therefore shorter jobs are more likely to be scheduled first.

2) SLA Violation Penalty of Batch Jobs: The second experiment compares the average penalty from our greedy algorithm and other methods for scheduling batch jobs. The algorithms we compared with are the Earliest Deadline First strategy (EDF) [14], the Least Slack Time First strategy (LST) [15],

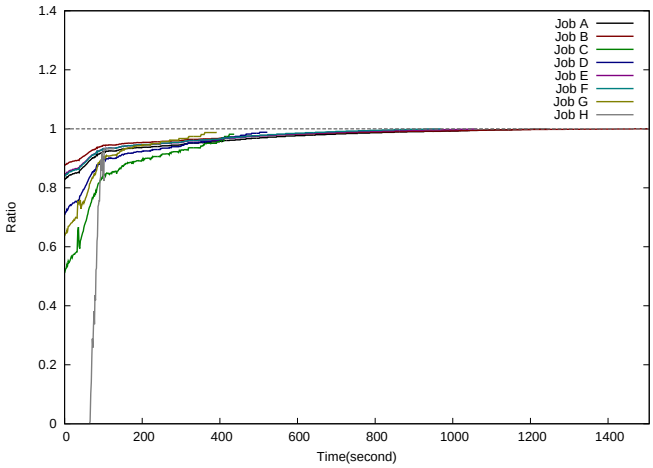

Fig. 4: Ratio between estimated penalty and final penalty of some batch jobs

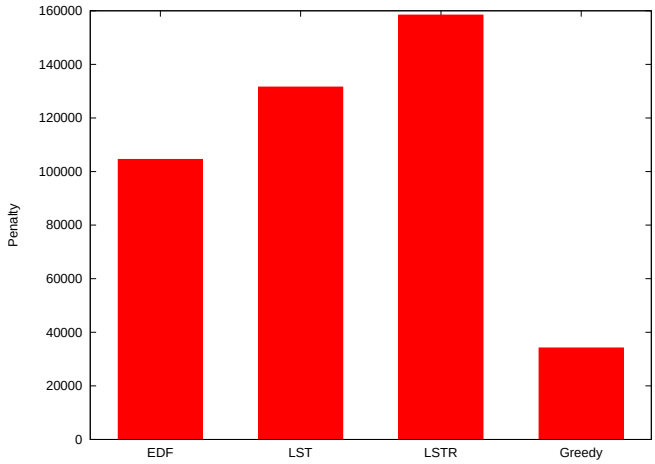

Fig. 6: Penalty of batch jobs with identical penalty rate

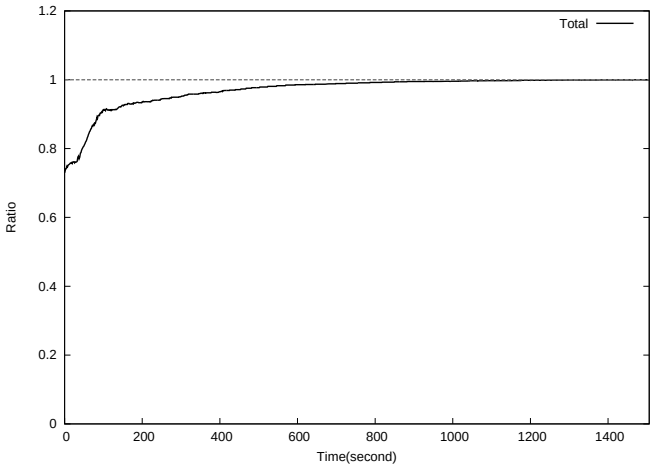

Fig. 5: Ratio between total estimated penalty and total final penalty of incomplete batch jobs

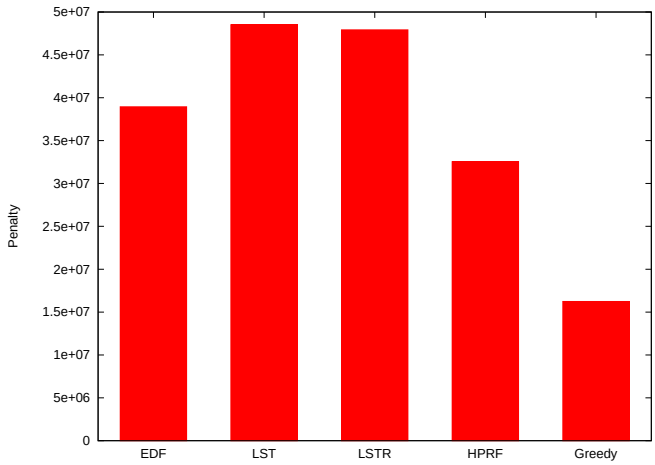

Fig. 7: Penalty of batch jobs with random penalty rate

the Least Slack Time Rate First strategy (LSTR) [16], and the Highest Penalty Rate First strategy (HPRF) algorithms.

We have two different settings in this experiment. In the first setting all jobs have *identical* penalty rates, 1. In the second setup, the penalty rates of jobs are randomly generated between 1 and 1000. For both settings we take 10 samples and schedule these samples separately. Then we observe the average penalty of the 10 runs. Figure 6 and Figure 7 show the average penalty. In the first setting when all jobs have identical penalty rates, the greedy algorithm produces 66.67% less penalty compared to the earliest deadline first algorithm. Note that in the first setting since all penalty rates are the same, the highest penalty rate first algorithm is not simulated. In the second setting the greedy algorithm produced 50% less penalty compared to the highest penalty rate first algorithm.

3) SLA Violation Penalty of Mixed Jobs: The third experiment compares the SLA violation penalty of our algorithm with the penalty from two other algorithms on mixed interactive and batch jobs. The first algorithms is a simple static fraction method, which reserves half of resource for batch jobs and the other half for interactive jobs. The second algorithm is a proportional method, which assigns processing units to interactive and batch jobs according to the ratio of their *recent penalty*.

The recent penalty is defined as follows. The recent penalty of batch jobs is the lower bound of the penalty of those incomplete batch jobs that have already missed their deadlines. Consequently it is estimated as the sum of individual penalty rate multiplied by the delay of all batch jobs. On the other hand, the recent penalty of interactive jobs is the sum of penalty incurred in the *previous* time window.

After determining the recent penalty from both batch and interactive jobs, the proportional method determines the number of processing units that should be assigned to batch jobs (m_b) and to interactive jobs (m_i) *proportionally* according to to their recent penalties. In case that the recent penalty of one type of job is insignificant compared to the other, the algorithm reserves 10% of the processing units for the type of jobs that has the smaller recent penalty.

For this mixed job experiment we select one sample of batch jobs from SDSC log and randomly set their penalty rate between 1 and 1000, and select five samples of Calgary-

HTTP log, and five samples of Saskatchewan-HTTP log as our interactive jobs. We set the penalty rate of interactive jobs to 10000, the fraction threshold to 0.9, and the time threshold of interactive jobs to twice the average request processing time. The size of a time window is 20 seconds. We submitted all interactive jobs and the first batch job at the beginning of the experiments. The rest of the batch jobs were submitted according to their submission time in the log. In the experiment, all interactive jobs lasted for one hour, and all batch jobs completed in less than 22 minutes.

Figure 8 shows the total SLA violation penalty from the dynamic programming method (using Equation 13) and the proportional method. For ease of discussion we will use DP to denote the dynamic programming method, SF to denote the static fraction method, and PP for the penalty proportion method.

We observe from Figure 8 that DP outperforms both SF and PP methods by having 19% less penalty. The batch job penalty of DP is 30% less than those of PP, and 46% less than those of SF. The interactive job penalty of DP is 10% less than those of PP, and 30% more than those of SF.

For ease of discussion on why DP outperforms PP, we plots the number of processing units allocated to a set of interactive jobs at different timestamps in Figure 9. Note that the rest of the processing units are allocated to batch jobs.

We believe that the DP algorithm outperforms the PP algorithm because it responds to job resource requirement *faster* than the PP algorithm does. From Figure 9 the DP algorithm allocated 54 processing units to batch jobs *as soon as* the experiment started. As most of the batch jobs completed, the number of processing units allocated to interactive jobs started to increase, and it increased to all processing units when all batch jobs finished.

In contrast, the PP algorithm responds to resource demand much slower that DP does. From Figure 9 at the beginning PP algorithm allocated half of processing units to batch jobs and the other half to interactive jobs because *none* of them has recent penalty. Soon after the second time window the PP algorithm shifted most of the processing units to interactive jobs because they started to have recent penalty. Meanwhile most batch jobs have not yet missed their deadlines, so they are not allocated enough processing units. As a result the PP algorithm only allocated 6 processing units, the minimum 10% guaranteed to either kind of jobs, to batch jobs from time 40 to 110. That is, the PP algorithm started to allocate more processing units to batch jobs *only* after some batch jobs missed their deadlines and the recent penalty of batch jobs started increasing, i.e., PP responded to the resource demand of batch jobs much *slower* than DP did. Consequently, the number of processing units allocated to batch jobs only gradually increased from 6 at time 110 to 58 at time 150. This delay in allocation, i.e., the number of processing units remains 6 from time 40 to 110, made more batch jobs miss their deadlines *afterwards*, and caused 40% more penalty than DP at the end. In addition, the penalty of interactive jobs of PP is 10% more than DP because less resource was allocated to them from 150 to 620 seconds.

The DP algorithm outperforms the SF algorithm because the SF algorithm does *not* consider the requirement of jobs.

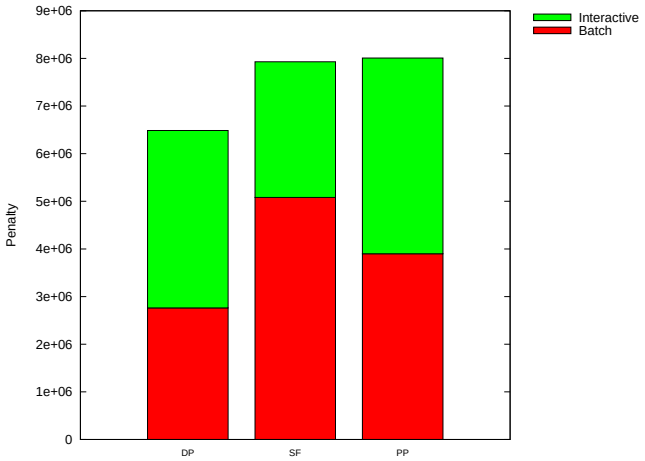

Fig. 8: Penalty of mixed jobs

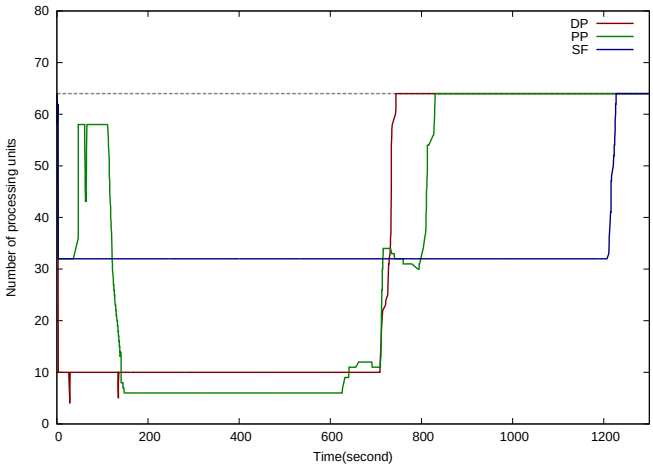

Fig. 9: Number of processing units allocated to interactive jobs

The SF algorithm statically allocated half of the processing units to batch jobs until all of batch jobs completed. Because SF allocated much fewer processing units to batch jobs than DP did, the penalty of batch jobs of SF is 83% more than those of DP. Since SF allocated at least half of processing units to interactive jobs all the time, the penalty of interactive jobs of SF is 24% less than those of DP. Nevertheless PP still outperform SF by having 19% less penalty.

VII. Conclusion

In this paper we described a scheduling framework that allocates resources to both batch jobs and interactive jobs simultaneously. We propose a model to formally quantify the penalty of both batch jobs and interactive jobs. Based on this model we also propose algorithms to estimate the penalty for both batch jobs and interactive jobs, and reduce the total SLA violation penalty. Our experiment results suggest that our system effectively reduces total penalty by allocating the right amount of resources to heterogeneous jobs.

There are many interesting future research directions. First, we would like to dynamically determine the best number of

processing units. The total number of processing units is fixed in our work. However, the total number of processing units can be determined based on the amount of different types of resources in the cluster and the requirements of jobs. For example, if most of the jobs are I/O-intensive, the total number of processing units should be based on the I/O bandwidth of machines, instead of the number of processor cores.

The time complexity of our greedy algorithm is $O(n^3)$ for n batch jobs. Even though we parallelize and optimized the algorithm, the computation could be time consuming for large number of batch jobs. We would like to explore the possibility to further optimize the algorithm or to design a new algorithm with similar concept but lower time complexity.

REFERENCES

[1] Michael Armbrust, O Fox, Rean Griffith, Anthony D Joseph, Y Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. M.: Above the clouds: a berkeley view of cloud computing. 2009.

[2] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. Ieee, 2010.

[3] Jian Li, Sen Su, Xiang Cheng, Meina Song, Liyu Ma, and Jie Wang. Cost-efficient coordinated scheduling for leasing cloud resources on hybrid workloads. *Parallel Computing*, 44:1–17, 2015.

[4] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[5] Saurabh Kumar Garg, Srinivasa K Gopalaiyengar, and Rajkumar Buyya. Sla-based resource provisioning for heterogeneous workloads in a virtualized cloud datacenter. In *Algorithms and Architectures for Parallel Processing*, pages 371–384. Springer, 2011.

[6] Martin T Hagan, Howard B Demuth, Mark H Beale, et al. *Neural network design*. Pws Pub. Boston, 1996.

[7] Chee Shin Yeo and Rajkumar Buyya. Service level agreement based allocation of cluster resources: Handling penalty to enhance utility. In *Cluster Computing, 2005. IEEE International*, pages 1–10. IEEE, 2005.

[8] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 195–204. IEEE, 2011.

[9] Raymond Keith Clark. *Scheduling dependent real-time activities*. PhD thesis, Carnegie Mellon University, 1990.

[10] Shuo Liu, Gang Quan, and Shangping Ren. On-line scheduling of real-time services for cloud computing. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 459–464. IEEE, 2010.

[11] Yue Yu, Shangping Ren, Nianen Chen, and Xing Wang. Profit and penalty aware (pp-aware) scheduling for tasks with variable task execution time. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 334–339. ACM, 2010.

[12] Hyeonjoong Cho, Binoy Ravindran, and E Douglas Jensen. On utility accrual processor scheduling with wait-free synchronization for embedded real-time software. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 918–922. ACM, 2006.

[13] R Garey Michael and S Johnson David. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, page 237, 1979.

[14] Michael L Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2012.

[15] Shrikant S Panwalkar and Wafik Iskander. A survey of scheduling rules. *Operations research*, 25(1):45–61, 1977.

[16] Myunggwon Hwang, Dongjin Choi, and Pankoo Kim. Least slack time rate first: New scheduling algorithm for multi-processor environment. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 806–811. IEEE, 2010.

[17] San diego supercomputer center. http://www.sdsc.edu/.

[18] D Feitelson, D TALBY, and JP JONES. Standard workload format, 2007.

[19] The internet traffic archive. http://ita.ee.lbl.gov/.