Building Semi-Elastic Virtual Clusters for Cost-Effective HPC Cloud Resource Provisioning

Shuangcheng Niu, Jidong Zhai, Xiaosong Ma, Xiongchao Tang, Wenguang Chen, and Weimin Zheng

Abstract—Recent studies have found cloud environments increasingly appealing for executing HPC applications, including tightly coupled parallel simulations. At the same time, while public clouds offer elastic, on-demand resource provisioning and pay-as-you-go pricing, individual users setting up their on-demand virtual clusters may not be able to take full advantage of common cost-saving opportunities, such as reserved instances. In this paper, we propose a *Semi-Elastic Cluster* (SEC) computing model for organizations to reserve and dynamically resize a virtual cloud-based cluster. We present a set of integrated batch scheduling plus resource scaling strategies uniquely enabled by SEC, as well as an online reserved instance provisioning algorithm based on job history. Our trace-driven simulation results show that such a model has a 61.0 percent cost saving than individual users acquiring and managing cloud resources without causing longer average job wait time. Moreover, to exploit the advantages of different public clouds, we also extend SEC to a multi-cloud environment, where SEC can get a lower cost than on any single cloud. We design and implement a prototype system of the SEC model and evaluate it in terms of management overhead and average job wait time. Experimental results show that the management overhead is negligible with respect to the job wait time.

Index Terms—Cloud computing, job scheduling, resource provisioning, semi-elastic cluster, trace-driven simulation

1 Introduction

With the introduction of HPC-oriented cloud platforms such as the Amazon EC2 Cluster Compute Instances (CCIs), people have been re-examining the feasibility of cloud-based parallel computing. While studies several years ago revealed an order-of-magnitude performance difference for tightly-coupled applications (such as MPI programs) between cloud and traditional platforms [1], recent work reported that CCIs have been closing on the gap and in many cases deliver comparable performances [2]. Given additional advantages in saving management overhead and having regular hardware upgrades and/or price drops, it may be financially and operationally wise for an HPC user to adopt the cloud as his/her major compute platform.

While cloud systems are not expected to replace state-of-the-art supercomputers in performing PetaFlop scale hero jobs using 10,000s or more processes, they might have become a practical alternative to typical medium- or large-scale clusters maintained by academic, research, or business organizations. In this paper, we examine such possibility by proposing a new model of organization-level cloud resource provisioning and management. Rather than owning physical, fixed-capacity clusters, organizations can reserve and dynamically resize a cloud-based virtual cluster, which can then be provided to its member users as a traditional parallel computer using the familiar batch scheduling interface. More specifically, we propose a Semi-Elastic Cluster (SEC) computing model, which allows organizations to maintain a set of cloud instances, schedule jobs within the current capacity, with the flexibility of dynamically adjusting the capacity level according to the current system load, user requirement in responsiveness (queue wait time), and the cloud provider's charging granularity.

While there are recent projects enabling shared cloud-based virtual clusters (such as StarCluster [3]) and exploring cost-saving approaches such as using EC2 spot instances [4], [5], our SEC model is unique in that it focuses on *workload aggregation*. This is motivated by the potential of utilizing the economies of scale, a major factor contributing to the success of public clouds themselves. Cloud resource providers such as Amazon give deep discount to heavy users through *reserved instances*, where users pay an upfront fee to reserve an instance for one-year or three-year terms, and enjoy different discounted hourly charge rates according to their expected usage level.

Fig. 1 illustrates a sample of dynamic SEC capacity variation in a period of five days simulated using a real workload 391-day trace collected from Data Star at the San Diego Supercomputer Center (SDSC) which has 1,640 processors [6] (All of data here are computed with offline greedy algorithm described in Section 5.1). It can be seen that the virtual cluster capacity adapted seamlessly to the actual system workload, varying dramatically between 2 and 200 instances (take Amazon EC2 cc2.8xlarge instances for example), minimizing resource idling inevitable with fixed-capacity clusters (the gap between the red boundary and the green fill).

- *S. Niu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and the Naval Aeronautic Engineering Institute, Yantai, China. E-mail: niu.shuangcheng@gmail.com.*
- *J. Zhai, X. Tang, W. Chen, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {zhaijidong, cwg, zwm-dcs}@tsinghua.edu.cn, tomxice@gmail.com.*
- *X. Ma is with the Qatar Computing Research Institute, Doha, Qatar. E-mail: xma@qf.org.qa.*

Manuscript received 28 Apr. 2015; revised 12 Aug. 2015; accepted 20 Aug. 2015. Date of publication 2 Sept. 2015; date of current version 15 June 2016. Recommended for acceptance by G. Tan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2476459

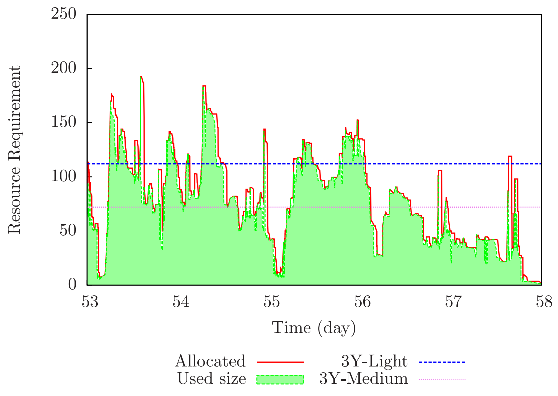


Fig. 1. A sample of resource usage variation at Data Star of SDSC (The green fill shows the real resource usage by cluster users. The red boundary shows dynamic SEC capacity variation.).

In Fig. 1, the dashed and dotted lines indicate the number of instances qualifying for using the 3Y-Light and 3Y-Medium reserved instances, respectively. It shows that the vast majority of resource usage can be performed on such deeply discounted instances, requiring the provisioning of on-demand instances to the rare cases when a large cluster is needed.

Table 1 further demonstrates the effectiveness of our SEC resource management scheme spanning all three types of instances. It compares the distribution of all the service units consumed (around 1.2 million instance-hour) on each pricing class. With the *Individual* mode, where each user individually acquires his/her virtual clusters and runs jobs, only one user can take advantage of the reserved instances, allowing one 3Y-Light instance altogether. As a result, over 99.8 percent of the total service units are spent on on-demand instances with the *Individual* mode. SEC, in contrast, effortlessly aggregates the workloads from concurrent users of the HPC center to enable the use of 72 3Y-Medium and 40 3Y-Light instances, reducing the on-demand consumption to less than 11 percent.

Our proposed SEC model exploits such economies of scale in three ways. First, it aggregates the demands from multiple users, enabling a "Groupon" mode in cloud computing for obtaining lower per-instance rates. Also, each reserved instance gets to be fully utilized after the organization covers the upfront reservation fee. Second, with the central batch queue and job submission history maintained through batch scheduling, SEC performs online prediction-based instance provisioning with different types of reserved instances to optimize its overall cost effectiveness. It can intelligently control the virtual cluster capacity and plan its resource distribution across different cloud pricing classes. Finally, sharing instances among multiple users allows an organization to efficiently utilize residual resources incurred by common cloud charging granularity, as well as to amortize the latency of booting cloud-based virtual clusters.

In the rest of the paper, we first illustrate the SEC model and present a set of integrated batch scheduling plus resource scaling strategies uniquely enabled by SEC. Our simulation experiment is based on EC2's CCI Eight Extra Large (cc2.8xlarge) instances using real HPC workload traces. The EC2 pricing scheme has changed quite some in the past years. Please note that the time of

TABLE 1
Resource Type Distribution Comparison

Instance type	3Y-Medium	3Y-Light	On-demand
SEC	73.66%	15.75%	10.59%
Individual	0	0.15%	99.85%

the pricing parameters used in this work is *06/2013*. The comparison result shows that a hybrid SEC has a 61.0 percent cost saving than individual cloud model, without increasing the average job wait time. A comparison with in-house clusters of different sizes shows that SEC systems provide an attractive resource provisioning option. Even not considering the extra benefits of scalability, covered management/maintenance, and continuous hardware/software upgrade, SEC offers a compromise balancing the cost and wait time concerns. We also evaluate SEC with a recent released public cloud, called Google Compute Engine (GCE) [7], which provides a fine-grained and flexible pricing model. Results show that SEC on GCE has a higher utilization than on EC2.

Next, we present a pair of offline and online algorithms to derive target instance counts for different pricing classes. The offline algorithm in this work is used to evaluate the optimization performance of the online approach. A major challenge we face lies in the validation of our proposed algorithms. As SEC is our envisioned new HPC computing paradigm, there are no existing workload traces, for us to verify our proposed approaches. Our intuition here is that if available, a SEC cluster may possess workload dynamics that sits between those on traditional supercomputers/clusters (where the workload growth is bounded by the fixed machine size) and on social media/networks (where there is a much lower "entry requirement" for users to start adopting a new tool or paradigm). Therefore, we evaluate our online load prediction and instance provisioning algorithms using traces from these two ends of the workload dynamics spectrum. The simulation results show that the performance of our proposed online method is very close to the offline method.

Third, in order to exploit the advantages of different cloud resource providers, we extend SEC to a multi-cloud environment, where SEC can get a lower cost than on any single cloud. We propose an online resource provisioning algorithm to request new instances from multi-cloud platforms. We evaluate our model on a multi-cloud platform using both Amazon EC2 and GCE clouds. Experimental results show that the multi-cloud model can reduce cost by 2.8-10.9 percent over single Amazon EC2 and single GCE.

Finally, to verify the feasibility of SEC, especially to assess the overhead of acquiring/maintaining shared cloud instances for batch job execution, we implemented a proof-of-concept SEC management framework based on SLURM [8], which is an open-source resource manager designed for Linux clusters. We then tested job submission, execution, and instance management on real Amazon EC2 resources. Based on our experimental results, we found the SEC overhead to be quite reasonable (at the seconds level). The overhead has negligible effect on the average job wait time and the average charge rate.

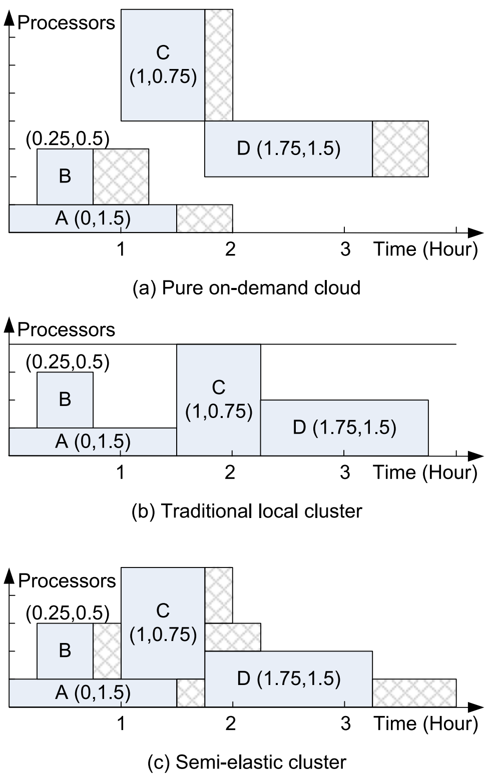


Fig. 2. Semi-elastic cluster model.

2 SEC Rationale

First, we illustrate the working of a SEC virtual cluster with a group of sample jobs, as shown in Fig. 2. Each job is portrayed as a rectangle, with the horizontal and vertical dimensions representing its resource requirement, in execution time and number of nodes needed. Each job is labeled with its (arrival time, execution time) pair. The gray boxes indicate the actual job execution periods on allocated instances/nodes. The shaded boxes indicate the unused *residual instance lease terms*, due to the common hourly charging granularity of public clouds.

With individual cloud execution in Fig. 2a, each job is submitted separately by a different user, who acquires/releases an individual virtual cluster. Hence each job is immediately allocated the requested resources upon arrival, leading to 0 wait time. The system utilization rate is 70.8 percent due to unused residual allocations. With traditional fixed-capacity four-node cluster in Fig. 2b, job C and D are delayed due to insufficient resources. This generates an average wait time of 15 minutes, with system utilization at 56.7 percent. With SEC in Fig. 2c, idle instances get reused by a different job (e.g., job C), resulting in system utilization at 77.3 percent. As the cluster size can dynamically adapt to the load, each job can be scheduled at its arrival, delivering the same wait time as in Fig. 2a.

Obviously, SEC wins over the traditional cluster mode by elasticity: reducing wait time through capacity growth and reducing cost through capacity reduction. Its advantage over the *individual* cloud execution mode in cost is also intuitive, even when this example does not demonstrate workload aggregation for discounted rates. However, our study finds that in terms of job wait time, SEC also outperforms the *individual* mode. The key observation is that with on-demand cloud instances, the actual wait time is significantly higher than 0. As can be seen from Table 2, it takes several minutes to boot an EC2 cluster before an MPI job can be launched, and the overhead grows as the cluster size increases. Such an overhead, in addition to the hourly charging granularity, makes cloud execution unappealing to short jobs, extremely common in the development and testing of parallel programs, where the instant execution enabled by cloud is highly desirable. SEC, on the other hand, is ideal for such workloads. By promoting resource aggregation and instance reuse, it removes such multi-fold penalties against development/debugging workloads: high start-up latency, wasted allocation between job runs, and light usage not qualifying for reserved instances.

TABLE 2
Average Instance Boot Time Measured when Requesting Different Numbers of Instances on Amazon CCI Platform (with Three Trials Each)

# of instances	Average boot time (minutes)
1	2.1
2	3.1
4	4.2
8	4.5
16	5.0

Note that SEC's advantage in workload aggregation and instance reuse persists even if cloud providers adjust their charging granularity (e.g., with minute-level usage accounting with an instance startup fee).

3 System Overview

Fig. 3 shows the architecture of SEC. Its main intelligence resides in SEC Management System (SMS), which manages a pool of computing nodes acquired from one or more public clouds. SMS processes job requests and controls the size of the virtual cluster according to its current status and the length of job queue. It improves the long-term cost-effectiveness of the virtual cluster by analyzing system logs and dynamically adjusting the reserved instance portfolio.

More specifically, SMS consists of three main components: a job scheduler, a resource manager, and a workload predictor. The job scheduler is responsible for receiving and scheduling job requests and making cost-effective job placement optimization. The resource manager coordinates with the job scheduler to manage the provisioning and release of cloud instances. It also records the status of SEC (including cluster size, per-pricing-class idle node counts, the number of nodes

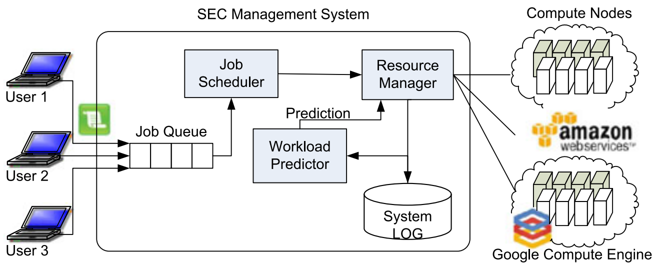


Fig. 3. Semi-Elastic Cluster system architecture.

in each reserved instance class and their lease start times, etc.) when the status changes (more details are illustrated in Section 7). In addition, it carries out a long-term planning to determine the purchase of reserved instances, working together with the workload predictor, which periodically analyzes system historical logs to predict future usage level. Such planning determines the amount of reserved instances to be purchased from each available instance class. Note that SMS itself can be run either on a dedicated local server or on a cloud instance. SEC is used solely on the users' side and does not need the cloud providers to change any current interfaces of clouds for adopting this model.

SEC is proposed for organizations to take the place of their own physical, fixed-capacity clusters. With such a model, organizations can reserve a virtual cluster and provide their users with fine-grained, lower pricing cloud instances than public clouds. Users no longer worry about the complex bidding strategies and can be charged as the conventional scheduling systems according to their usage.

4 Integrated Cluster Size Scaling and Batch Job Scheduling

With our proposed SEC model, a cloud-based cluster for executing batch parallel jobs can expand or shrink according to the current system load. Traditional batch scheduling algorithms, designed for fixed-capacity parallel machines, have to be extended in this scenario. In addition to prioritizing and dispatching jobs, it also has to perform resource provisioning, in the form of dynamic cluster size scaling.

Our discussion is based on priority-based backfilling algorithms, the common job scheduling strategy adopted by current HPC systems. With such algorithms, the priority of each job is dynamically calculated (typically considering factors such as the user-specified priority level, time in the queue, and job size in terms of the number of nodes/instances requested), by which the jobs are sorted in the job queue. Resource reservation will be made for one or several jobs at the top of the queue (*top jobs*), based on the expected execution time specified in the job script. Smaller jobs, by their priority, may have a chance to be *backfilled* into the "holes" left by such reservation.

4.1 Dynamic Cloud Instance Provisioning

SEC enables the capacity of a virtual cluster to grow or shrink, by dynamically acquiring/releasing cloud instances. Such a provisioning-enabled algorithm faces the tradeoff between responsiveness (average job wait time) and cost (overall monetary cost of executing a batch of parallel jobs). To this end, we present and evaluate a strategy that attempts to minimize the cost under a configurable wait time constraint.

Instance acquisition. Checking for instance acquisition need is triggered by the same events that prompt checking the possibility of dispatching a new job: upon job arrival or completion. Only these events can affect the queue status or the expected dispatch time of waiting jobs. After regular scheduling procedures like job dispatch, priority updates, backfilling, and queue update, new instances will be requested iff

- the top job's size exceeds the current virtual cluster capacity, or
- the top job's predicted wait time exceeds a threshold ($T_{waitlmt}$).

Here $T_{waitlmt}$ is a tunable parameter between 0 and 1 hour (EC2's billing granularity). Intuitively, choosing a $T_{waitlmt}$ value beyond the billing granularity increases the average job wait time without further reducing the monetary cost. In our experiments, we set it at 5 minutes (according to Table 2), to match the latency of setting up one's individual cloud cluster.

In the case of cluster expansion, the SEC scheduler needs to determine how many new instances to request from the cloud. In our design, we explored three strategies in deciding the amount of resources to request, with different levels of look-ahead in the job queue (such as requesting only the instances needed by the top job or requesting instances needed by all the jobs in the queue).

- *FirstSize*, calculated according to the resource requirement of the top job. With this strategy, low-priority jobs' wait time is not bounded by $T_{waitlmt}$, as it may take several dispatch operations for them to move to the top of the queue. However, our relatively low $T_{waitlmt}$ value renders a short job queue most of the time, so only a small number of jobs will be affected.
- *SumSize*, the sum of resource requirement from all waiting jobs. In this case, all waiting jobs will be guaranteed a wait time bounded by $T_{waitlmt}$, obviously at the cost of a larger cluster footprint (and lower utilization).
- *BestSize*, an optimized allocation size calculated based on the current waiting job information, which can be viewed as a hybrid approach between *FirstSize* and *SumSize*. It separates short jobs from long jobs in the queue, using a tunable threshold T_{short}, then calculate the total number of instances to request as the sum of *SumSize* for all long jobs and *FirstSize* for short jobs. The intuition is that short jobs have a higher chance of executing sequentially by utilizing the residual allocations from existing instances, based on the observation that short jobs are abundant at HPC centers. In fact, *FirstSize* and *SumSize* can be seen as the special case of *BestSize*.

Instance release. Unlike the acquisition, instance release is checked periodically (every one minute in our experiments) after job scheduling. Considering Amazon's hourly pricing model, it makes more financial sense to perform *delayed release*: an idle instance will be freed only at the end of its hourly allocation term, when it is about to "expire". Such periodic release check is skipped when the job queue is not empty (in this case, all idle nodes are marked "reserved" for a waiting job). Upon each release check, all idle instances with a residual lease time shorter than the checking frequency will be released. Note that when a SEC system has multiple classes of reserved instances, it is not important to select the more expensive instances to release. As to be discussed in more details later, cloud providers such as EC2 automatically charge instance owners in a way that maximize the utilization of the cheaper reserved instances.

4.2 Job Placement

Finally, when making reservations for a waiting job or dispatching a newly arrived job, our scheduling algorithm needs to address the SEC-unique job placement problem. We consider several placement strategies:

- *Random*, where instances are selected randomly from the idle instance pool,
- *MaxMarginFirst*, where instances are selected by their residual lease term in descending order to maximize the utilizing residual lease time,
- *MinMarginFirst*, where instances are selected by their residual lease term in ascending order to minimize the utilizing residual lease time,
- *MaxIdleFirst*, where instances are selected by their current idle duration in descending order to minimize the utilization level of instances,
- *MinIdleFirst*, where instances are selected by their current idle duration in ascending order to maximize the utilization level of instances.

5 Automatic Configuration of Reserved Instance Portfolio

One of the motivation for SEC is to take advantage of economies of scale by aggregating workloads from users, whose individual usage level may not qualify for discounted hourly rates available only to heavy users. For example, Amazon EC2 has different reserved instance classes for long-term users, such as “3-Year Light”, “3-Year Medium”, and “3-Year Heavy”. As mentioned earlier, those reserved instance classes have different upfront fees and discounted rates, each hitting the sweet point of a certain long-term usage level. The more heavily discounted instance classes are connected with longer term of resource usage commitment, typically purchased when users are confident about their (heavy) consumption behavior in the next few years. As purchased instances cannot be returned, there are certain risks associated with reserving such instances.

Intuitively, a SEC system needs to maintain multiple types of instances to increase its flexibility, reduce the risk of purchasing heavy-usage instances, and optimize its overall cost-effectiveness. The rationale here is that a “guaranteed” minimum level of aggregated usage can be satisfied with the most heavily discounted instance class, supplemented with multiple classes of more expensive yet less-committed classes. However, it is not obvious exactly how it should diversify its instance “portfolio”.

Fortunately, SEC is managed by an organization, where the aggregated job history can be recorded and analyzed. We argue that although individual users' *short-term* workload is rather unpredictable, SEC's load level aggregated from a large number of users is relatively stable, enabling long-term load level prediction based on job history. In the rest of this Section, we present a pair of offline and online algorithms to derive target instance counts for different pricing classes. The offline algorithm can be used to evaluate the optimization performance of the online approach.

5.1 Offline Reserved Instance Configuration

Reserved instances have lower hourly charge rates, but require non-trivial upfront reservation fees. It lowers rental cost only when its overall usage time is beyond a *minimum utilization level*. Given the upfront charge and discounted hourly rate, it is straightforward to calculate such minimum utilization level. For example, for Amazon EC2 cc2.8xlarge, a 3-Year Light instance has a minimum utilization level of 6.8 percent, while 3-Year Medium has 38.3 percent.

On Amazon cloud platform, when a user reserves a certain number of instances, he/she will automatically be charged in discounted hourly rates from the reserved instances, starting from the most heavily discounted. For example, when a user reserves 10 3-year heavy instances and 5 3-year light instances, he/she will be charged for all the hours during the 3-year lease at the 3-year heavy rate, and at the 3-year light rate for the 11th to the 15th instances running concurrently with the first 10 3-year heavy instances. Any concurrent usage beyond 15 instances will be billed as on-demand instances.

Based on such pricing structure, we first propose an offline algorithm to identify the optimal reserved instance provisioning plan for a given job history, produced by the combination of the job submission history and our SEC job scheduling algorithm described earlier in Section 4. As it will be shown in our evaluation results, the seemingly insignificant wait time, controlled by the parameter $T_{waitlmt}$, serves as a “load smoother” that can significantly reduce the burst from instance demands. Note that our scheduling algorithm incorporates dynamic instance provisioning (when and how many instances are acquired/released), but is independent of the reserved instance composition of a SEC cluster.

5.1.1 Configuration Problem Definition

Now we have, for a fixed workload trace, a capacity trace generated from our batch job scheduling algorithm. Below we formulate the global optimal instance provisioning problem.

The input is an $n \times m$ matrix U, where $U_{i,j} \in \{0,1\}$ is the utilization level of the jth instance during the ith instance charging interval, n is the total number of intervals, and m is the maximum cluster capacity. Note that each vertical vector U_i has values monotonically decreasing from bottom to up.

The cloud provides pricing classes $\{C_0, C_1, \ldots C_h\}$, where C_0 is on-demand and C_h is the highest discount class. Each pricing class can be denoted as a tuple $\langle T_k, F_k, P_k \rangle$, representing its reservation term, upfront fee, and hourly price.

The optimal solution to the problem is an $n \times h$ pricing class matrix R, where $R_{i,k} \geq 0$ is the number of reserved instances of class k purchased at the ith charging interval, so that the total cost of running the given workload trace is minimized.

The total rental cost has a complicated computational process. Firstly, the number of active reserved instances of the kth pricing class at the ith moment is

$$S_{i,k} = \sum_{l=i-T_k/\tau+1}^{i} R_{l,k}. \tag{1}$$

Therefore, the hourly charge rate of jth instance at the ith moment ($U_{i,j}$) is

$$Rate_{i,j} = \begin{cases} P_h & \text{if } S_{i,h} \leq j, \\ P_l & \text{if } \sum_{k=l+1}^{h} S_{i,k} < j \leq \sum_{k=l}^{h} S_{i,k}, \\ \ldots & \ldots \\ P_0 & \text{other else.} \end{cases} \tag{2}$$

So the total rental cost (included upfront fee and hourly cost) is

$$Cost_{total} = Cost_{upfront} + Cost_{hourly} = \sum_{i=1}^{n}\sum_{k=1}^{h}(R_{i,k} \cdot F_k) + \sum_{i=1}^{n}\sum_{j=1}^{m}(U_{i,j} \cdot Rate_{i,j}).$$

We believe the problem to be difficult, though we have not been able to prove it NP-hard, or to find a polynomial-time solution. However, we tackle this unknown complexity by proposing a pair of offline algorithms: one sub-optimal greedy solution, plus one optimal-competitive algorithm delivering better-than-optimal solution. The true global optimal cost should be between the results given by these two algorithms, which are used as references to evaluate our online provisioning algorithm proposed later.

5.1.2 Offline Greedy Algorithm

We propose a forward offline greedy algorithm to optimize instance reservation. The basic assumption is that at the beginning of each time interval, we look ahead at the utilization matrix U. The reservation action only occurs at the beginning of an interval. To reduce the computation granularity, we choose a larger time interval, e.g., a week, and consolidate the hourly usage per instance within each such interval (so $U_{i,j}$ may be greater than 1). We list the main steps of the greedy algorithm at below (A formal description of this algorithm is listed in our preliminary version [9]):

- Step 1: Calculate, row by row, the jth instance's utilization level during different reservation terms (e.g., 1Y, 3Y). With this result, we iterate over the pricing classes C, starting from the most discounted class C_h, and identify the first class whose minimum utilization level is met by the jth instance. After an examining for the upcoming interval, we get a provisioning plan containing each instance's pricing class assignment.
- Step 2: Check the inventory of reserved instances. We exclude expired reserved instances. Then, all active reserved instances are re-divided according to their expiration time (e.g., active 3Y-Heavy instances with remaining lease under one year will be classified as 1Y-Heavy instances).
- Step 3: Compare the provisioning plan with the current inventory and decide the purchase amount. The purchase decision is first set to minimize the difference between the sum of new inventories and the sum of desired plans with Equation (3) which has a group of solutions, and then minimize the difference between the new inventory and the desired plan of each reserved instance class with Equation (4) from these solutions:

$$\min\left|\sum_{k=1}^{h}(S_{i,k} + R_{i,k} - D_{i,k})\right|, \tag{3}$$

$$\min\sum_{k=1}^{h}\left|S_{i,k} + R_{i,k} - D_{i,k}\right|, \tag{4}$$

where $R_{i,k} \geq 0$, $S_{i,k}$ is the inventory of reserved instances C_k at the ith time interval and $D_{i,k}$ is the desired provisioning plan.

5.1.3 Offline Optimal-Competitive Algorithm

Next, we propose an optimal-competitive algorithm which has a better-than-optimal total cost. The difficulty of the original provisioning problem lies in the fact that its reserved instance reservation period (1Y/3Y) differs with the reservation decision interval (e.g., weekly). To eliminate this hurdle, we transform the original cloud pricing classes C into new classes C', whose reservation period length matches that of the decision interval τ, with upfront fee scaled down accordingly. For example, for a pricing class $C_k =< T_k, F_k, P_k >$, its transferred class is $C'_k =< \tau, F_k \cdot \tau/T_k, P_k >$. In Section 9, we give a formal proof on C''s cost competitiveness.

SEC generally maintains an inventory of reserved instances at the end of trace-based simulation. We also define *pure cost* by deducting the residual inventory value, calculated by prorating the total cost over the remaining unused lease. We can prove that the same conclusion also applies to the pure cost.

With C', constructing the instance portfolio is rather trivial. Its minimal total cost can be calculated as follows:

$$\sum_{ij}\min_{k}\left(F_k \cdot \tau/T_k + P_k \cdot U_{i,j}\right). \tag{5}$$

The intuition here is that with the interval of making decision matching the period of reservation, one can individually decide the best instance to use for the jth instance at the ith interval, with no penalty associated with switching between instance classes and these small intervals.

5.2 Online Reserved Instance Configuration

While offline optimization is shown to be expensive at least in the above discussion, real SEC managers have to perform *online* instance provisioning without seeing a full job history. In particular, reserved instances are relatively long-term commitments (one year and three years in EC2's case) and cannot be "returned", demanding a load prediction mechanism that sees rather far into the future. In comparison, prediction-based resource management proposed in prior study focused on estimating what will happen in the next seconds or minutes [10], [11], [12].

We recognize that (1) the effectiveness of long-term time series prediction is limited (e.g., the lack of accurate stock trend prediction) and (2) building white-box, analytical model for SEC usage is challenging as it depends on individual SEC cluster's resources and policies, community demand, competing facilities, and collective user behavior. In this paper, we propose an online algorithm that strives to make a reasonable long-term SEC usage trend prediction (Section 5.2.1), to be coupled with conservative instance reservation strategies (Section 5.2.2).

A major challenge we face lies in the validation of our proposed algorithms. As SEC is our envisioned new HPC computing paradigm, there are no existing workload traces, not to mention multi-year traces, for us to verify our proposed approaches. Our intuition here is that if available, a SEC cluster may possess workload dynamics that sits between those on traditional supercomputers/clusters (where the workload growth is bounded by the fixed machine size) and on social media/networks (where there is a much lower "entry requirement" for users to start adopting a new tool or paradigm). In Section 8, we evaluate our online load prediction and instance provisioning algorithms using traces from these two ends of the workload dynamics spectrum.

5.2.1 Long-Term Instance Demand Prediction

We aim to predict the total number of instances needed in a future time interval. To reduce both computation complexity and short-term variance that should have no impact on long-term reservation decisions, we use weekly time intervals.

We adopt classic Exponential Smoothing (ES), an important prediction technique for time series data. It is relatively simple but quite robust for processing non-stationary noises [13], [14], and is widely used in business to forecast demands for inventories [14]. In contrast to simple moving average methods, ES takes into account *all* past points, rather than just the past k. Meanwhile, it performs surprisingly well in forecasting compared to more sophisticated approaches [15], [16].

More specifically, we extend classical Holt's double-parameter ES method [14] to model the total number of active instances needed at a given future time interval. We assume that the evolution of active instances is a continuous, smooth curve that can be fitted with a quadratic polynomial:

$$m_{k+t} = d_k + v_k t + a_k t^2/2. \tag{6}$$

Here, m_{k+t} is the instance demand level at time $k+t$, d_k is the estimated demand level at time k, v_k is the estimated rate of demand change (trend), a_k is the estimated acceleration (change rate of trend). These time-varying parameters can be estimated with a triple-parameter ES:

$$\begin{aligned} d_k &= \alpha m_k + (1-\alpha)(d_{k-1} + v_{k-1} + a_{k-1}/2), \\ v_k &= \beta(d_k - d_{k-1}) + (1-\beta)(v_{k-1} + a_{k-1}), \\ a_k &= \gamma(v_k - v_{k-1}) + (1-\gamma)a_{k-1}, \end{aligned} \tag{7}$$

α, β and γ are smoothing factors, with values between 0 to 1, that determine the relative weight given to recent changes versus historical data ("responsive" versus "smoothing") [14]. In this work, we extend the widely-used dynamic smoothing factor estimation method proposed by Trigg and Leach [17], by adjusting the smoothing factor values according to the observed prediction errors.

Finally, we perform instance demand forecast for the wth week in the future using Equation (6), replacing t with w.

5.2.2 Prediction-Based Instance Provisioning

With the weekly instance demand projected as discussed earlier, a SEC cluster can make dynamic decisions on whether to make any new reservations at the beginning of each weekly time interval. More specifically, it makes adjustment to its reserved instance inventory using a greedy algorithm by the following steps below:

- Step 1: Update the current reserved instance inventory, by removing reservations that have expired during the past week.
- Step 2: Perform weekly instance demand prediction for the next M months, which is equal to the longest reservation period provided by the cloud service (three years in EC2's case), using the triple ES algorithm described in Section 5.2.1.
- Step 3: Apply the offline instance provisioning optimization algorithm discussed in Section 5.1 on the projected weekly demand. This calculates the requirement for new reservations.
- Step 4: Make reservations according to the result of Step 3, and update the reserved instance inventory.

As the ES prediction algorithm keeps adjusting its prediction according to new usage history data points, the above instance provisioning algorithm possesses limited self-correcting capability. We are more concerned with over-estimated future demands, which lead to overly aggressive instance reservation. It is possible to extend the basic algorithm above, so that SEC managers can configure their cloud-based clusters with different risk-control policies, such as preference toward shorter-term reservations, upper limit on the number of reserved instances from certain pricing classes, etc.

6 Optimization for Multi-Cloud Platforms

The pricing models of various cloud resource providers are always different. For example, Google Compute Engine provides a fine-grained pricing model where all instances are charged by a minimum of 10 minute cost. After 10 minutes, instances are charged in 1 minute. At the same time, GCE automatically gives a discount for every incremental minute usage of each instance. The more the instance is used, the more discount. Moreover, all the instances do not need to be ordered in advance. Therefore, this model is more beneficial for short-term requirements. In contrast, as mentioned before, Amazon EC2 provides a course-grained pricing model and charges instances hourly, but it provides reserved instances considering long-term requirements.

In order to exploit the advantages of different cloud platforms, we extend the SEC model to a multi-cloud environment, where SEC can get a lower cost than on any single cloud platform. On such a multi-cloud platform, a main challenge is that which cloud platform should be chosen for requesting new instances to run the waiting jobs, because the requested instances on one cloud platform are difficult to be migrated to another one due to large migration overhead. Here, we propose a greedy selection algorithm to request new instances by the following steps:

- Step 1: Construct a requirement list of resources from the waiting job queue according to the instance acquisition strategies, such as *FirstSize*, *SumSize*, and *BestSize* which are described in Section 4.1.
- Step 2: Select a cost-effective object cloud platform for each requirement in the above list. SEC needs to predict the average charging rate of different cloud platforms for current requirement. We use the instant charging rate for the prediction when requesting new instances. The prediction needs to

consider the current number of requested instances, reserved instance configurations, and the usage history of instances, according to the specific pricing model of each cloud platform.

- Step 3: Consolidate the requirements and request resources from the object cloud platform. Note that the quantity of the requested resources need to subtract the number of idle resources.

The algorithm above is invoked when new instances need to be requested. Note that when configuring the reserved instance portfolio, we extend the online prediction-based instance provisioning algorithm presented in Section 5.2 by adding the GCE pricing model to support the multi-cloud environment.

Similarly, when deploying a newly arrived job, we also use a greedy algorithm to choose an object cloud platform and then use the same strategies described in Section 4.2 for job placement.

7 Implementation

We implement a prototype of SEC system and evaluate it using both Amazon EC2 and GCE. The scheduling system of SEC is implemented on top of SLURM [8], an open-source resource manager designed for HPC clusters. The resource management policies discussed in previous sections are implemented as a plugin of SLURM.

The prototype system is event-driven. After initialization, the system gathers the information of computing nodes, including host name, the start time of provisioning, billing cycle, node status, IP address etc. When a job arrives or finishes, it synchronizes the states of computing nodes and schedules waiting jobs. When the condition of resource expanding is matched, the system requests new computing nodes from public clouds. When the condition of releasing nodes is matched, the system releases idle nodes. When new nodes are allocated or idle nodes are released, the system re-configures the cluster and broadcasts the changes to all nodes. All the state changes of above system are logged. The system predicts reserved instance demand based on the history at every weekend. When the condition is matched or existing reserved instance expires, it purchases new reserved instances.

The system performs job scheduling and resource scaling using the algorithms in Sections 4 and 5. It has a number of internal parameters for effective SEC management (such as instance provisioning and job placement strategies, as well as threshold values). The job scheduler is based on the classic EASY backfilling scheduling algorithm [18].

For the convenience of end-users, the system sets user accounts and pre-configures a Network File System (NFS) [19]. When new nodes are allocated, the system configures them. The private program and data of users can be stored in cloud storages, such as S3. Temporary data can be stored in each node. When jobs finish, the temporary data is wiped off to protect users' privacy.

8 Evaluation

8.1 Experiment Setup

Cloud platforms. We use Amazon's EC2 Cluster Compute Instances and Google Compute Engine to evaluate our SEC scheduling and resource provisioning (through simulation). On Amazon EC2, our simulation is based on the CCI Eight Extra Large (cc2.8xlarge) Instances, each node has 16 processors(2 x Intel Xeon E5-2670, eight-core), 60.5 GiB memory and 3,370 GB instance storage. In calculating job wait time on EC2, we include the cc2.8xlarge boot time according to our actual EC2 measurement at the appropriate requested resource size (partially listed in Table 2). On GCE, all cloud instance types are charged in 1 minute, after a 10-minute initial charging. GCE automatically gives a charging discount according to instances' utilization level on a monthly basis. Compared to EC2's CCI, GCE provides a much more flexible pricing model for short jobs, while it does not provide any discounted reserved instances.

Workload traces. As mentioned earlier, SEC clusters do not yet exist so there are no workload traces available that fit the proposed elastic plus batch scheduling profile. We evaluate our proposed techniques using HPC job traces (with fixed cluster size) and social network user population traces (with low adoption barrier).

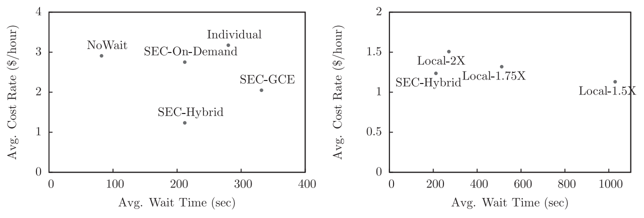
From the HPC side, we use several real workload traces collected from production systems [6]. The job entries are traditional HPC jobs, usually tightly-coupled parallel applications (such as MPI programs). For each job, the trace entry contains attributes such as the requested number of nodes, user-estimated and actual execution time, submission time stamp, wait time, user ID, etc. For evaluating our SEC scheduling policies and overall cost-effectiveness, we used a 391-day trace from Data Star, a 184 node, 1,640-processor IBM 655/690 system at San Diego Supercomputer Center. It contains 96,089 jobs submitted between Mar. 2004 and Mar. 2005. For evaluating our online instance reservation optimization, we used six workload traces with longer duration time from the same collection. They include, besides the same Data Star trace, workload history from SDSC's Blue Horizon (SDSC Blue), SDSC's IBM SP2 (SDSC SP2), Cornell Theory Center IBM SP2 (CTC SP2), High-Performance Computing Center North (HPC2N), and Sandia Ross cluster(Sandia Ross).

From the social network side, we use the SNS name search traffic from Google Trends [20] as our workloads. We suspect that the search traffic closely correlates to the active user population size for such services, based on Twitter's two-year active user population size [21], [22]. For this study, we select six well-known services: Facebook, Twitter, Yelp, MySpace, Flickr and Renren (a Chinese social networking site). These traces contain normalized weekly search traffic, with length ranging between four and eight years.

Metrics. We evaluate our system with three metrics, include that average job wait time, overall system utilization, and monetary cost. The first two are standard metrics for schedulers, while the third is calculated as the overall cloud instance rental cost for SEC clusters. When using reserved instances, total costs of SEC clusters also include that reservation costs. However, SEC generally maintains an inventory of reserved instances at the end of trace-based simulation. To be fair, we calculate a pure cost that deduct inventory values in accordance with remaining expire period.

8.2 Single-Cloud Cost-Responsiveness Analysis

Intuitively, with workload aggregation, SEC lowers cost but incurs higher wait time. However, on-demand cloud cluster



(a) SEC vs. individual on-demand cloud execution (b) SEC vs. owning local clusters

Fig. 4. Cost-responsiveness evaluation.

provisioning is not latency-free, as we have shown with the multi-minute instance boot delay. SEC enables instance reuse and can control the wait time through the $T_{waitlmt}$ parameter.

In this section, we examine the cost-responsiveness tradeoff in adopting SEC and compare it with alternative modes of cloud HPC. More specifically, we simulated the following modes: 1) *Individual*, as described earlier, 2) *NoWait*, a naive workload aggregation mode, where the cluster immediately requests on-demand resources whenever an arriving job does not have enough nodes to start, practically eliminating the job queue, 3) two variations of SEC on Amazon EC2, with *SEC-On-Demand* using only on-demand instances and *SEC-Hybrid* using all reserved instances classes and on-demand instances, 4) *SEC-GCE*, SEC using instances of GCE. Here, we only list the result of GCE with the same hourly price as EC2 for the same instance class. More results on GCE are presented in Section 8.6. For each mode, we replayed the Data Star trace and calculated wait time and cost rate.

Fig. 4a shows the performance of all execution modes, in both cost (in terms of the average hourly rate per node-hour) and the average job wait time. Over *Individual*, *SEC-On-Demand* obtains a 13.3 percent , *SEC-Hybrid* obtains a 61.0 percent, and *SEC-GCE* obtains a 35.4 percent cost saving. Meanwhile, both SEC modes on EC2 actually *reduce* the average job wait time of *Individual* by 24.2 percent, through instance reuse. On GCE, due to its fine-grained pricing model, *SEC-GCE* obtains the highest instance utilization level (98.4 percent) among all these modes. At the same time, it leads to a 18.4 percent increasing of the job wait time over *Individual* due to the more frequent instance releasing with the fine-grained model. *NoWait* has significantly lower average job wait time, at around 82 seconds, due to its combination of instant job dispatch and instance reuse. Meanwhile, its cost is much closer to *Individual* than to *SEC-Hybrid*.

Finally, we compare the cost-effectiveness and responsiveness of SEC with those of owning a traditional, fixed-capacity cluster. Based on the largest job from the Data Star trace (requesting 1,480 processes, translating to 93 16-core cc2.8xlarge instances), we consider multiple candidate capacity settings, at sizes 93 (1×), 117 (1.25×), 140 (1.5×), 163 (1.75×), and 186 (2×), respectively.

In estimating the in-house cluster cost (hardware and operational), we used a similar approach as in our prior work [2], amortizing the hardware/hosting cost over a three year cluster lifetime and prorate the cost in executing the Data Star trace. Table 3 lists the common expense items, assuming similar configuration as the cc2.8xlarge instances. While this estimate adopts InfiniBand interconnection (the common configuration for today's small- or mid-sized clusters), it does not include any management personnel budget, a non-trivial item with clusters of such sizes.

TABLE 3
In-House Cluster Expense Items

Expense item	Price	#
Computing Nodes (DELL PowerEdge R720)	$7,749	93
InfiniBand NIC	$612	93
InfiniBand Switch (36 ports)	$9,172	3
I/O Nodes (DELL PowerEdge R720)	$7,749	5
Disk Array (HP D2600 with 24 TB disks)	$6,399	10
Fiber NIC for I/O	$1,043	98
Fiber Channel Switch (16 ports)	$6,599	8
Hosting for one year (energy included)	$15,251	3
Total Cost (3 years)	$1,108,583	

Cost comparison between cloud-based and traditional HPC resources depends on the relative performance of the same application across the two platforms. In this paper, we assume a 1:1 ratio, assuming that a cloud cluster delivers the same performance as similarly equipped local clusters, which is shown to be the case for the majority of real-world applications we tested [2]. Fig. 4b shows the results, where the *Local-*$x\times$ clusters clearly demonstrate the tradeoff between cost and responsiveness. *SEC-Hybrid* has an hourly rate roughly equivalent to *Local-1.75* ×, but an average job wait time lower than even *Local-2* ×. *Local-1.5* × has a lower hourly rate than *SEC-Hybrid*, but possesses a wait time 4.8 times higher.

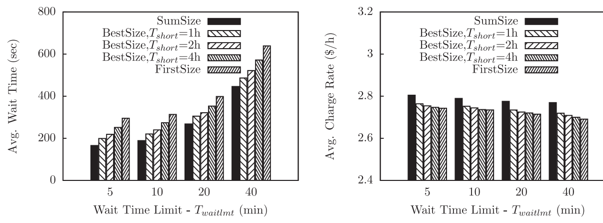
Note that in addition to the cost-responsiveness advantages, cloud resource provisioning is very different from buying hardware and provides users with much more flexibility. For example, cluster owners need to buy all hardware at once according to expected system load and job sizes, or risk complexity in performance and system management introduced by heterogeneity with system expansion/upgrades. Cloud instances can be reserved at any time and a virtual cluster of reasonable size can easily remain homogeneous.

8.3 SEC Scheduling Parameters Analysis

Our proposed job scheduling and resource provisioning algorithms in Section 4 use three important parameters/policies: (1) the top-queue job wait time threshold $T_{waitlmt}$, which controls the eagerness in growing the cluster size, (2) the short job threshold T_{short}, which controls the size of such growth (*SumSize* and *FirstSize* can be seen as special cases of *BestSize*, with $T_{short} = 0$ or ∞), and (3) the job placement policies, which determine allocated virtual machines to run jobs. In this Section, we evaluate the influence of these parameters/polices. To isolate such influence, our experiments here only use on-demand instances, which leads to higher hourly charge rates than typical SEC results shown in other figures.

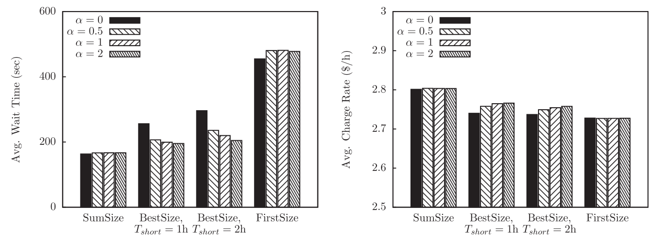
We first analyze the impact of job placement policies, and found that *MaxMarginFirst* offers the lowest cost on account of maximally utilizing the residual lease time of instances, while *MinMarginFirst* excels at cutting the average job wait time because the wasted instances serve as a resource buffer. The other three policies perform very close to each other and provide a compromise between the two requirements. More details are shown in our preliminary work [9].

Next, we analyze the impact of wait time threshold and short job threshold. Fig. 5 presents simulation results using



(a) Average wait time (b) Average charge rate

Fig. 5. Impact of scheduling and provisioning parameters.



(a) Average wait time (b) Average charge rate

Fig. 6. Impact of user-provided job runtime estimation.

the same job traces, with job placement policy set to *MaxMarginFirst*. It shows the average wait time with different parameter configurations. Fig. 5a shows that the average wait time has positive correlation with $T_{waitlmt}$ and T_{short}. In particular, $T_{waitlmt}$ has more significant influence and appears to be a straightforward tuning knob for desired responsiveness.

Fig. 5b shows the same parameters' impact on the average cost rate, with which both parameters have negative correlation. Here T_{short} appears to have more influence. With a higher T_{short} value, more low-priority short jobs wait in the queue and have a better chance to be scheduled by backfilling. This reduces cost by improving the overall system utilization and reduces wasted resources introduced by overestimated job execution time.

8.4 Impact of User-Provided Job Runtime Estimation Analysis

The classic backfilling algorithm relies on the runtime estimate of jobs, which is provided by users. However, statistics show that users are very likely to provide a much longer runtime estimate than its real execution time [23], [24]. In this Section, we evaluate how SEC makes a decision under different degrees of overestimation. To do quantitative analysis, we use a similar approach as in our previous work [25]. We introduce a parameter α, where $t_{sch} = t_{run} + \alpha \times (t_{req} - t_{run})$. Here t_{req} is the job's user requested run time, t_{run} is the job's real run time in the trace. We use the t_{sch} as the job's user estimated runtime to submit to the simulator. In this manner, we can obtain workload traces with different overestimation degrees by tuning the α. When $\alpha = 0$, $t_{sch} = t_{run}$ and when $\alpha = 1$, $t_{sch} = t_{req}$. The larger α is, the more inaccurate estimate time will be.

In Fig. 6, we simulate the cases when α is 0, 0.5, 1 and 2 using the Data Star traces, with the two scheduling policies fixed ($T_{waitlmt} = 5$ min, placement policy = MaxMarginFirst). Fig. 6a shows the impact on average job wait time, and Fig. 6b shows the impact on average charge rate.

These results show that user-provided estimate time has different effects depending on the expanding policy. With *SumSize*, the expanding size is independent of the user estimation. With *BestSize*, the SEC requests new resources based on user estimation. A more accurate estimate time will reduce expanding size, thereby lower average charge rate and increase average job wait time. On the contrary, with *FirstSize*, an accurate estimate time will slightly decrease the average job wait time. The major difference with classical in-house cluster is that the overestimation has a negligible effect on system utilization (less than 0.95 percent), that is reflected with average charging rate in Fig. 6.

8.5 Reserved Instance Configuration Methods Analysis

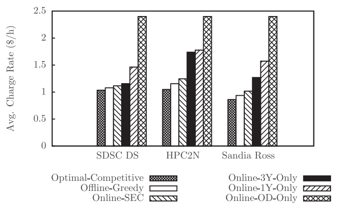
Next, we examine our dynamic instance reservation algorithms through trace-driven simulation. Without loss of generality, we choose several representative pricing classes from EC2 (for CCI cc2.8xlarge instances), namely "3Y-Heavy", "3Y-Light", "1Y-Heavy", "1Y-Light", and "On-demand". Here we mainly focus on the dynamics in overall resource usage and concurrency requirement (how many instances are needed simultaneously), rather than individual jobs' execution.

As mentioned earlier, we use non-elastic HPC workloads and highly-dynamic SNS workloads to "bracket" potential SEC workload dynamics, in validating our reserved instance provisioning algorithms. The HPC workload time-series data are derived by driving our SEC scheduler ($T_{waitlmt} = 5$ min, $T_{short} = 1$ hour, placement policy = *MaxMarginFirst*) with real HPC job traces. The SNS workloads are synthesis workloads, generated by "spreading" the total instance usage from the DataStar trace according to each social network service's name search traffic from Google Trends [20]. This is based on the assumption that the total resource usage is strongly correlated to user population, which is in turn strongly correlated to the Google search traffic. To test the effectiveness of our proposed long-term resource demand prediction, we intentionally selected services with steady growth (Twitter and Yelp), slowed-down growth (Facebook), and growth followed by steady decline (MySpace, Flickr, and Renren) [9].

Note that the actual job mix (in terms of length, sizes, and perhaps other non-functional parameters) of SNS-like workloads might be very different than in the HPC workloads. The SNS data in this paper is only used to evaluate our proposed online method. Moreover, the prediction of this paper is a long-term and coarse-grained online load prediction which is different from the utilization prediction in most work about the cloud literature [11], [12].

In addition to the methods proposed in Section 5, To assess the advantage of SEC resource provisioning using a diverse instance portfolio, we compare our proposed online method (*Online-SEC*) with online methods using restricted instance classes: on-demand (*Online-OD-Only*), 1-year reserved (*Online-1Y-Only*), and 3-year reserved (*Online-3Y-Only*).

Fig. 7a shows simulation results with HPC traces. We have analyzed six HPC workloads. Among of them, SDSC Blue, SDSC SP2 and CTC SP2 have a similar performance with the aforementioned SDSC Data Star (SDSC DS) trace, so we do not include them here. Fig. 7a shows that the performance of our proposed online method is very close to


(a) Rental cost w.o. inventory using HPC traces

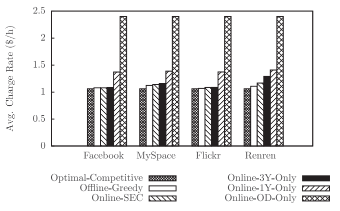

(b) Rental cost w.o. inventory using SNS traces

Fig. 7. Average charge rate using different algorithms.

both offline methods. Moreover, it also shows that our method is better than the methods with only a single type of instance. Fig. 7b presents simulation results for SNS data, offering similar observations.

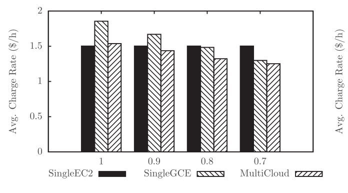
From the above simulation results, we can conclude that our *Online-SEC* method has the following benefits: (1) it eliminates the short-term load level fluctuations through a smoothing method, thereby avoiding the shortage or waste of reserved instances, and (2) it effectively provisions multiple instances classes according to a long-term trends prediction, thus obtaining a trade-off between scaling and cost efficiency.

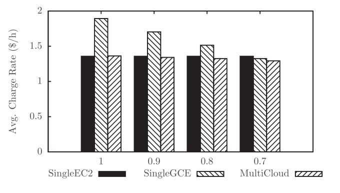
8.6 Multi-Cloud Cost-Responsiveness Analysis

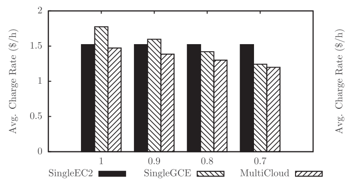
In this Section, we examine our SEC model on a multi-cloud platform combined with Amazon EC2 and Google Compute Engine. We also compare the multi-cloud cost with two single-cloud results, Amazon EC2 and GCE (denoted by SingleEC2 and SingleGCE on Fig. 8). For EC2, we choose several representative pricing classes for cc2.8xlarge instances: 3Y-Heavy, 3Y-Light, 1Y-Heavy, 1Y-Light and on-demand. A challenge is that these cloud platforms have different instance classes with different computing performance and different hourly prices. Moreover, their prices are adjusted continuously. To facilitate analysis, we assume that GCE has the same instance class with the cc2.8xlarge instance of EC2. The difference is that they do not have the same hourly price. We introduce a parameter $r = price\_GCE/price\_EC2$ to describe the relative ratio between them for effective evaluation.

Since GCE does not provide any reserved instance for discount prices, there is not any inventory values at the end of simulations. Although EC2 provides an online marketplace [26] that gives customers some flexibility to sell their unused reserved instances, selling off all the unused reserved instances is not an easy thing. To be fair, we evaluate our model with the total monetary cost including residual inventory values. We also optimize EC2 reserved instance configurations with total cost using our online methods described in Section 5.2.

In Fig. 8, we simulate the cases when r is 1, 0.9, 0.8, and 0.7 using HPC workloads, with our SEC scheduler ($T_{waitlmt} = 5$ minutes, $T_{short} = 1$ hour, placement policy=*MaxMarginFirst*). Due to the limited space, we only present the simulation results of four traces. The results show that GCE has a roughly equivalent average charging rate with EC2 with single cloud platform when r is 0.8. This also means that both pricing models have a comparative total cost when r is 0.8. The results also show that our SEC model on the multi-cloud platform can save 2.8-10.9 percent cost


(a) HPC2N


(b) SDSC Blue


(c) Sandia Ross

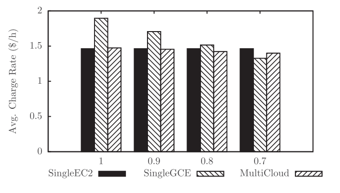

(d) SDSC SP2

Fig. 8. The average charging rate of the SEC model on multi-cloud and single-cloud platforms with different price ratios (r) between GCE and EC2.

compared with on single EC2 or GCE when r is equal to 0.8. However, when two single cloud platforms have relatively large cost difference (r is 1.0 or 0.7), there is a smaller saving or even tiny cost increasing, because requested instances cannot be migrated between different cloud platforms.

8.7 Overhead Analysis with SEC Prototype

To verify the feasibility of SEC, especially to assess the overhead of acquiring/maintaining shared cloud instances for batch job execution, we test job submission, execution, and instance management on real Amazon EC2 resources, and found the SEC overhead is quite reasonable.

An important issue associated with SEC is data protection/isolation with instance reuse. Per-user data include application codes, input/output data, and other temporal data. To create an environment similar to traditional parallel platforms, our SEC prototype manages separate user accounts on persistent cloud storage. Such data only can be accessed by the owner, protected by the cloud security mechanisms. Also, we leverage EC2's free per-instance ephemeral disks to provide users with "node-local" storage, where user data are not supposed to persist between job executions. While data on ephemeral disks do not persist between instance rental sessions, SEC reuses instances across different users' jobs. In our implementation, we take a conservative approach that formats the local storage when switching between jobs.

To this end, we experiment with reformatting EC2 ephemeral disks and find that file system type has a dominant impact on this overhead. E.g., each EC2 845 GB ephemeral disk takes, on average, 185 seconds to format with ext3, but only 3.25 seconds with ext4. With four ephemeral disks that come with cc2.8xlarge instances, it takes around 3.4 seconds to format all, including the unmount and remount overhead.

To run parallel jobs in the cloud, instance configuration tasks include configuring host names, updating hosts file, and configuring the file system (e.g., NFS). In addition, unique to SEC clusters, we need to set up user accounts and add nodes to the SLURM partition. We measure the total and SEC-incurred configuration time many times, requesting different numbers of instances, at different times of the day. Table 4 below gives the average measurement. Note that releasing instances, from both EC2 and SLURM, takes

TABLE 4
Average EC2 CCI Instance Set Configuration Time (Seconds)

# of instances	Total time	SEC-only time
1	7.36	2.11
2	7.46	2.14
4	7.76	2.22
8	8.26	2.36
16	9.25	2.65

around 5 seconds per instance. However, such time is not billed and not visible to users.

8.8 Impact of Overhead Analysis

Finally, we perform job scheduling simulation using the DataStar traces with varying aforementioned job scheduling parameters/policies to check the impact of the measured cluster management overhead.

Fig. 9 shows the differences of simulation results between with management overhead and without overhead by varying two scheduling parameters ($T_{waitlmt}$ and T_{short}), with placement policy fixed (MaxMarginFirst). As expected, the management overhead has negligible effects on average job waiting time and average charge rate.

Fig. 9a shows the impact on average job waiting time. Fig. 9b shows the impact on average charge rate. From the results, we can conclude that in most cases, the management overhead has a negligible effect on average job wait time (which is about 0.5-4.9 percent), and brings a slightly higher charge rate (about 0.02-0.37 percent). In some special cases (e.g., $T_{waitlmt} = 5$ min, expanding policy=SumSize), it has a lower job wait time and a higher charge rate. This is because that the management overhead delays the release of resources, thereby increasing average charge cost. Meanwhile, the idle instances which are not released as a resource pool, reduce average job wait time.

9 Proof of the Offline Optimal-Competitive Algorithm

Theorem 1. *The minimal rental cost using pricing classes C' is no more than the optimal solution using pricing classes C for a given utilization matrix $U_{n\times m}$.*

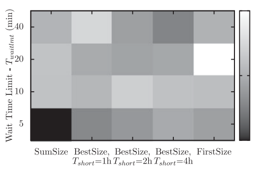
Suppose the minimal rental cost using C occurs with solution provisioning matrix $R = \{R_{i,k}\}_{n\times h}$. The number of active reserved instances of the kth pricing class at the ith moment is

$$S_{i,k} = \sum_{l=i-T_k/\tau+1}^{i} R_{l,k}. \tag{8}$$

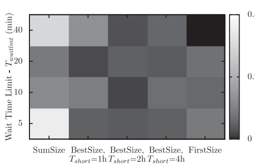
Therefore, the hourly charge rate of jth instance at the ith moment ($U_{i,j}$) is

$$Rate_{i,j} = \begin{cases} P_h & \text{if } S_{i,h} \le j, \\ P_l & \text{if } \sum_{k=l+1}^{h} S_{i,k} < j \le \sum_{k=l}^{h} S_{i,k}, \\ \dots & \dots \\ P_0 & \text{other else.} \end{cases} \tag{9}$$

The total rental cost consists of two parts: upfront fee and hourly cost. So, the total cost using pricing classes C is



(a) Average wait time



(b) Average charge rate

Fig. 9. Impact of SEC management overhead.

$$\begin{aligned} Cost_{total} &= Cost_{upfront} + Cost_{hourly} \\ &= \sum_{i=1}^{n}\sum_{k=1}^{h}(R_{i,k}\cdot F_k) + \sum_{i=1}^{n}\sum_{j=1}^{m}(U_{i,j}\cdot Rate_{i,j}). \end{aligned}$$

Now, we construct a solution $R' = \{R'_{i,k}\}_{n\times h}$ using pricing classes C', where $R'_{i,k} = S_{i,k}$. For example, with the decision making window decreased to one week, we replicate the original solution R by renting the same number of C'_k instances in the same week (such as replacing a yearly rental with 52 serial weekly rental starting at the same time). Therefore, C' and C have the same number of active reserved instances from the kth pricing class during each C' decision making window:

$$S'_{i,k} = R'_{i,k} = S_{i,k}, \forall i \forall k.$$

Therefore, C' and C have the same hourly charge cost

$$Rate'_{i,j} = Rate_{i,j}, \forall i \forall j => Cost'_{hourly} = Cost_{hourly}. \tag{10}$$

$\sum_{i=1}^{n} R_{i,k}$ means the sum of requirements of C_k from the first to the nth moment. While $\sum_{i=1}^{n} S_{i,k}$ stands for the sum of inventories. A purchased reserved instance C_k will be valid during next T_k time, and will be counted multi times (T_k/τ) in the sum of inventories. If all purchased reserved instances are expired at the final moment, then $\sum_{i=1}^{n} R_{i,k} = \frac{\tau}{T_k}\sum_{i=1}^{n} S_{i,k}$. However, in most cases, there are some residual inventory at the final moment. Therefore,

$$\sum_{i=1}^{n} R_{i,k} \ge \frac{\tau}{T_k}\sum_{i=1}^{n} S_{i,k} = \frac{\tau}{T_k}\sum_{i=1}^{n} R'_{i,k}, \forall k. \tag{11}$$

The difference in total cost between pricing classes C and C' is

$$\begin{aligned} \Delta_{total} &= \Delta_{upfront} \\ &= \sum_{i=1}^{n}\sum_{k=1}^{h}(R_{i,k}\cdot F_k) - \sum_{i=1}^{n}\sum_{k=1}^{h}\left(R'_{i,k}\cdot F_k \cdot \frac{\tau}{T_k}\right) \\ &= \sum_{k=1}^{h}\left(F_k \cdot \left(\sum_{i=1}^{n} R_{i,k} - \frac{\tau}{T_k}\sum_{i=1}^{n} R'_{i,k}\right)\right) \ge 0. \end{aligned}$$

From above derivations, we construct a solution R' using pricing classes C' whose rental cost is no more than the optimal solution R using pricing class C. Therefore, we conclude that the minimal rental cost using pricing classes C' is no more than the optimal solution using pricing class C for a given utilization matrix $U_{n\times m}$.

10 Related Work

10.1 Virtual Clusters

Recently, some projects are focusing on facilitating cloud computing [3], [27], [28], [29]. For example, StarCluster [3] can help users easily create a virtual cluster on Amazon's EC2 platform. However, StarCluster does not support dynamic resource adjustment based on history workload and it also cannot perform load aggregation from different users to purchase discounted reserved instances. Liu and He [29] proposed a novel resource allocation mechanism, called reciprocal resource fairness, to address the economic fairness issue for multiple resource types in public clouds.

Several scheduling algorithms for cloud-based virtual clusters have been proposed. Moschakis et al. explored the usage of two common Gang Scheduling strategies [30]. Genaud and Gossa evaluated the strategies based on classic scheduling and bin-packing algorithms with Amazon's pricing model [31]. Although these studies consider the trade-off between cost and response time, they have not combined batch scheduling algorithms with dynamic cloud resource scaling.

10.2 Cost-Effective Cloud Computing

Many researchers have studied the cost minimization on public cloud platforms. Some of them focused on Amazon EC2 spot instances. For example, Andrzejak et al. proposed a probabilistic model to help users to bid resources [32]. While Javadi et al. used a statistical model to fit spot prices and the inter-price time (time between price changes) [4]. Yi et al. investigated checkpointing mechanisms to minimize costs while maximizing the reliability with spot instances [5]. Marathe et al. exploited the redundancy on spot instances and designed an adaptive algorithm to help users to select scheduling algorithms and determine the optimal bid price [33]. Zhao et al. studied how to minimize cost for elastic applications while meeting application service requirements [34]. These work can help users make informed bids, improve the reliability, and predict the spot prices.

There are some studies focusing on Amazon reserved instances. Shen et al. proposed a heuristics-based policy that minimizes the cost by combining reserved instances with on-demand instances under an assumption that there exists a perfect predictor [35]. Wang et al. proposed two online algorithms by limiting the discussion to one type of reserved instances based on rough estimations [36]. Our proposed SEC model is a beneficial complement for these work through load aggregation and instances reuse.

10.3 Prediction Algorithms in Computer Systems

A lot of time-series based prediction methods for performance parameters or resource demand in computer systems have been proposed [10], [11], [12], [37], [38]. Zhai et al. proposed a replay-based performance prediction method for large-scale parallel applications [37]. Zheng [38] and Kalyvianaki [10] adopted Kalman filter to track changes for performance parameters of a web application and processor usage level in a data center. Duy et al. proposed a neural network predictor to optimize server power consumption in cloud computing [11]. Ardagna et al. used a simple Exponential Smoothing to predict jobs arrival rate in cloud platform [12].

11 Conclusion

We presented SEC, a new execution model for HPC in the cloud that manages variable-size clusters to be shared by many users within an organization. This model brings new roles to traditional batch job scheduling algorithms, by incorporating resource provisioning and management problems into parallel scheduling. Our trace-driven simulation results show that SEC can deliver much more cost-effective cloud scheduling than individual users acquiring and managing cloud resources without causing longer average job wait time. Moreover, to exploit the advantages of different public clouds, we also extend SEC to a multi-cloud environment, where SEC can get a much lower cost than on any single cloud.

Acknowledgments

A preliminary version of this work has been published in SC'13[9]. The authors sincerely thank the anonymous reviewers for their valuable comments and suggestions. This work has been partly supported by the Chinese National High-Tech Research and Development Plan (863 project) 2012AA01A302, NSFC projects 61232008 and 61472201, as well as Research Fund for Taishan Scholar Project of Shandong Province. It has also been sponsored by the US National Science Foundation (NSF) grants 0915861, 0937908, 0958311, and 1318564, in addition to a joint faculty appointment between ORNL and NCSU, plus a senior visiting scholarship at Tsinghua University. Jidong Zhai is the corresponding author.

References

[1] E. Walker, "Benchmarking amazon EC2 for high-performance scientific computing," *Login*, vol. 33, no. 5, pp. 18–23, 2008.

[2] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus in-house cluster: Evaluating amazon cluster compute instances for running MPI applications," in *Proc. State Practice Rep.*, 2011, p. 11.

[3] (2012). StarCluster Online. Available: http://web.mit.edu/star/cluster/

[4] B. Javadi, R. Thulasiramy, and R. Buyya, "Statistical modeling of spot instance prices in public cloud environments," in *Proc. IEEE 4th Int. Conf. Utility Cloud Comput.*, 2011, pp. 219–228.

[5] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the Amazon elastic compute cloud," in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, 2010, pp. 236–243.

[6] (2012). Parallel Workloads Archive Online. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[7] (2014). G. Inc, Google Compute Engine Pricing Online. Available: https://cloud.google.com/compute/pricing

[8] (2012). SLURM: A Highly Scalable Resource Manager, Online. Available: https://computing.llnl.gov/linux/slurm/

[9] S. Niu, J. Zhai, X. Ma, X. Tang, and W. Chen, "Cost-effective cloud HPC resource provisioning by building semi-elastic virtual clusters," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. 1–12.

[10] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters," in *Proc. 6th Int. Conf. Autonomic Comput.*, 2009, pp. 117–126.

[11] T. V. T. Duy, Y. Sato, and Y. Inoguchi, "Performance evaluation of a green scheduling algorithm for energy savings in cloud computing," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum*, 2010, pp. 1–8.

[12] D. Ardagna, S. Casolari, and B. Panicucci, "Flexible distributed capacity allocation and load redirect algorithms for cloud systems," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 163–170.

[13] B. Billah, M. L. King, R. D. Snyder, and A. B. Koehler, "Exponential smoothing model selection for forecasting," *Int. J. Forecasting*, vol. 22, no. 2, pp. 239–247, 2006.

[14] E. S. Gardner Jr, "Exponential smoothing: The state of the art," *J. Forecasting*, vol. 4, no. 1, pp. 1–28, 1985.
[15] S. Makridakis, A. Andersen, R. Carbone, R. Fildes, M. Hibon, R. Lewandowski, J. Newton, E. Parzen, and R. Winkler, "The accuracy of extrapolation (time series) methods: Results of a forecasting competition," *J. Forecasting*, vol. 1, no. 2, pp. 111–153, 1982.
[16] S. Makridakis and M. Hibon, "The M3-competition: Results, conclusions and implications," *Int. J. Forecasting*, vol. 16, no. 4, pp. 451–476, 2000.
[17] D. Trigg and A. Leach, "Exponential smoothing with an adaptive response rate," *Oper. Res. Soc.*, vol. 18, pp. 53–59, 1967.
[18] D. Lifka, "The ANL/IBM SP scheduling system," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 1995, pp. 295–303.
[19] B. Nowicki, *NFS: Network File System Protocol Specification*, Network Working Group RFC1094, 1989.
[20] (2013). G. Inc., Google Trends [Online]. Available: http://www.google.com/trends/
[21] C. Alex and E. Mark. (2009). An in-depth look inside the twitter world [Online]. Available: http://blog.rjmetrics.com/new-data-on-twitters-users-and-engagement/
[22] J. Stein. (2010). New Data on Twitter's Users and Engagement [Online]. Available: http://blog.rjmetrics.com/new-data-on-twitters-users-and-engagement/
[23] D. Feitelson and A. Weil, "Utilization and predictability in scheduling the IBM SP2 with backfilling," in *Proc. 1st Merged Int. Symp. Parallel Distrib. Process. Parallel Process. Symp.*, 1998, pp. 542–546.
[24] C. Bailey Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are user runtime estimates inherently inaccurate?" in *Proc. 10th Int. Workshop Job Scheduling Strategies Parallel Process.*, 2005, pp. 253–263.
[25] S. Niu, J. Zhai, X. Ma, M. Liu, Y. Zhai, W. Chen, and W. Zheng, "Employing checkpoint to improve job scheduling in large-scale systems," in *Proc. 16th Int. Workshop Job Scheduling Strategies Parallel Process.*, 2013, pp. 36–55.
[26] A. Inc., (2012). Amazon EC2 Reserved Instance Marketplace Online. Available: http://aws.amazon.com/ec2/reserved-instances/marketplace/
[27] W. Voorsluys, S. Garg, and R. Buyya, "Provisioning spot market cloud resources to create cost-effective virtual clusters," in *Proc. 11th Int. Conf. Algorithms Archit. Parallel Process.*, 2011, pp. 395–408.
[28] M. Caballer, C. De Alfonso, F. Alvarruiz, and G. Moltó, "EC3: Elastic cloud computing cluster," *J. Comput. Syst. Sci.*, vol. 79, no. 8, pp. 1341–1351, 2013.
[29] H. Liu and B. He, "Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in IaaS clouds," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 970–981.
[30] I. Moschakis and H. Karatza, "Evaluation of gang scheduling performance and cost in a cloud computing system," *J. Supercomput.*, vol. 59, no. 2, pp. 975–992, 2012.
[31] S. Genaud and J. Gossa, "Cost-wait trade-offs in client-side resource provisioning with elastic clouds," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 1–8.
[32] A. Andrzejak, D. Kondo, and S. Yi, "Decision model for cloud computing under SLA constraints," in *Proc. IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2010, pp. 257–266.
[33] A. Marathe, R. Harris, D. Lowenthal, B. R. de Supinski, B. Rountree, and M. Schulz, "Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on amazon EC2," in *Proc. 23rd Int. Symp. High-Perfor. Parallel Distrib. Comput.*, 2014, pp. 279–290.
[34] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang, "Optimal resource rental planning for elastic applications in cloud market," in *Proc. 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 808–819.
[35] S. Shen, K. Deng, A. Iosup, and D. Epema, "Scheduling jobs in the cloud using on-demand and reserved instances," in *Proc. 19th Int. Conf. Euro-Par Parallel Process.*, 2013, pp. 242–254.
[36] W. Wang, B. Li, and B. Liang, "To reserve or not to reserve: Optimal online multi-instance acquisition in IaaS clouds," *arXiv preprint arXiv:1305.5608*, 2013.
[37] J. Zhai, W. Chen, and W. Zheng, "Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 305–314, 2010.
[38] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with extended Kalman filters," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.* , 2005, pp. 334–345.

Shuangcheng Niu received the BS and MS degrees from Naval University of Engineering of China in 1997 and 2000, respectively, and the PhD degree in computer science from Tsinghua University in 2013. He is currently an assistant professor in the Department of Scientific Research, Naval Aeronautical Engineering Institute, China. His research interests include high-performance computing and cloud computing.

Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China in 2003, and the PhD degree in computer science from Tsinghua University in 2010. He is currently an assistant professor in the Department of Computer Science and Technology, Tsinghua University. His research interests include high-performance computing, compilers and performance analysis, and optimization of parallel applications.

Xiaosong Ma received the BS degree in computer science from Peking University, China, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2003. She is currently a senior scientist at Qatar Computing Research Institute and an adjunct associate professor in the Department of Computer Science, North Carolina State University. Her research interests include the areas of cloud computing, storage systems, parallel I/O, and workload characterization.

Xiongchao Tang received the bachelor's degree from Tsinghua University in 2013, where he is currently working toward the PhD degree in the Institute of High Performance Computing. His major research interests include performance evaluation for high-performance computers, cloud computing, and fault tolerance of high-performance computers.

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. He was the CTO in Opportunity International Inc. from 2000 to 2002. Since January 2003, he has joined Tsinghua University. He is currently a professor and an associate head in the Department of Computer Science and Technology, Tsinghua University. His research interest is in parallel and distributed computing and programming model.

Weimin Zheng received the BS and MS degrees from Tsinghua University in 1970 and 1982, respectively. He is currently a professor in the Department of Computer Science and Technology, Tsinghua University. He is currently the director in China Computer Federation (CCF). His research interests include parallel and distributed computing, compiler technique, grid computing, and network storage.