Cost-effective Cloud HPC Resource Provisioning by Building Semi-Elastic Virtual Clusters

Shuangcheng Niu [1,2]
nsc07@mails.thu.edu.cn

Jidong Zhai [1]
zhaijidong@thu.edu.cn

Xiaosong Ma [3,4]
ma@csc.ncsu.edu

Xiongchao Tang [1]
tomxice@gmail.com

Wenguang Chen [1,2]
cwg@thu.edu.cn

[1] Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China
[2] Research Institute of Tsinghua University in Shenzhen, Shenzhen, China
[3] Department of Computer Science, North Carolina State University, USA
[4] Computer Science and Mathematics, Oak Ridge National Laboratory, USA

ABSTRACT

Recent studies have found cloud environments increasingly appealing for executing HPC applications, including tightly coupled parallel simulations. 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% cost saving than individual users acquiring and managing cloud resources without causing longer average job wait time. Meanwhile, the overhead of acquiring/maintaining shared cloud instances is shown to take only a few seconds.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Cloud computing

General Terms

Management

Keywords

Cloud computing, resource provisioning, job scheduling

1. INTRODUCTION

SC'13, November 17-21, 2013, Denver, CO, USA

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 [34], recent work reported that CCIs have been closing on the gap and in many cases deliver comparable performances [36]. 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 [20,35], 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 1-year or 3-year terms, and enjoy

different discounted hourly charge rates according to their expected usage level.

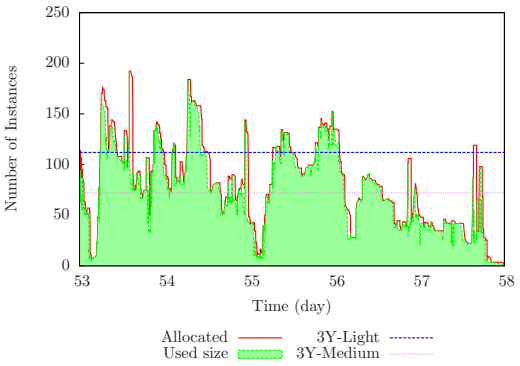


Figure 1: Sample SEC capacity variation

Figure 1 illustrates a sample of dynamic SEC capacity variation in a period of 5 days simulated with using a real workload 391-day trace collected from Data Star at the San Diego Supercomputer Center (SDSC) which has 1640 processors [1] (All of data here are computed with offline greedy algorithm described in Section 4.1). 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).

In Figure 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: 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%

Table 1 further demonstrates the effectiveness of our hybrid 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 acquire his/her virtual clusters and run jobs, only one heavy users can take advantage of the reserved instances, allowing one 3Y-Light instance altogether. As a result, over 99.8% 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%.

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 comparison result shows that a hybrid SEC has a 61.0% cost saving than individual cloud model, meanwhile its average job waiting time doesn't surpass individual cloud model. 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 between the cost and wait time concerns.

Next, 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. 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). 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 offline methods.

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 [2], 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 experiment results, we found the SEC overhead to be quite reasonable (at the seconds level). The overhead has negligible effect on the average job waiting time (0.1-2.4%).

2. SEC RATIONALE

First, we illustrate the working of a SEC virtual cluster with a group of sample jobs, as shown in Figure 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

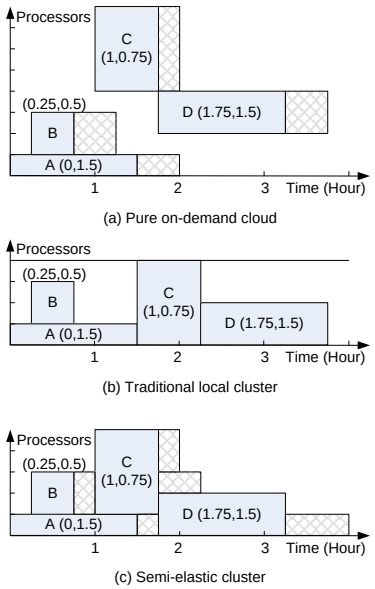


Figure 2: Semi-elastic cluster model

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 2(a), 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 its arrival, producing a logical job wait time of 0. The system utilization rate is 70.8% due to unused residual allocations. With traditional fixed-capacity 4-node cluster in 2(b), job C and D are delayed due to insufficient resources. This generates an average wait time of 15 minutes, with the system utilization at 56.7%. With SEC in 2(c), idle instances get reused by a different job (e.g., job C), resulting in a system utilization level of 77.3%. As the cluster size can dynamically adapt to the load, each job can be scheduled at its arrival time, delivering the same wait time as in 2(a).

Table 2: Average instance boot time measured when requesting different numbers of instances on Amazon CCI platform (with 3 trials each)

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

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.

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

3. 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, has 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.

3.1 Dynamic Cloud Instance Provisioning

SEC enables the capacity of a virtual cluster to grow or shrink, by dynamically acquiring/releasing cloud instances. Apparently such a provisioning-enabled scheduling algorithm faces the trade-off between responsiveness (job average waiting time) and cost (overall monetary cost of executing a batch of parallel jobs). In this paper, 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 the checking for the possibility of dispatching a new job: upon job arrival or completion. Only these events can affect the queue status and the expected dispatch time of waiting jobs. After regular scheduling procedures like job dispatch, priority updates, backfilling, and queue adjustment, 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 to 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 user experience of setting up one's individual cloud clusters.

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 for 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 performed except when the job queue is not empty (which implies that 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.

3.2 Job Placement

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

- *Random*, where instances are selected randomly from the idle instance pool,

- *MaxMarginFirst*, where instances are selected in the descending order of their residual lease term (reminiscent of the "best-fit" algorithm for memory allocation),

- *MinMarginFirst*, where instances are selected in the ascending order of their residual lease term (reminiscent of "worst-fit"),

- *MaxIdleFirst*, where instances are selected in the descending order of their current idle period duration, and

- *MinIdleFirst*, where instances are selected in the ascending order of their current idle period duration.

4. 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 a 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.

4.1 Offline Reserved Instance Configuration

Reserved instances have lower hourly charge rates, but also require non-trivial upfront reservation fees. Only when

its overall usage time is beyond a *minimum utilization level*, reserved instance will get a lower total rental cost. Given the upfront charge and discounted hourly rate, it is straightforward to calculate such minimum utilization level. For example, for Amazon EC2 cc2.8xlarge, its 3-Year Light instances have a minimum utilization level of 6.8%, while 3-Year Medium instances have 38.3%.

In Amazon cloud platform, when a user reserves a certain number of instances, he/she will automatically have at most that many instances in *concurrent* use covered by 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 3. 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.

4.1.1 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 (hour), n is the total number of intervals, and m is the maximum capacity of the SEC cluster (maximum number of concurrent instances needed). Note that each vertical vector U_i has value monotonically decreasing from bottom to up.

The cloud provides pricing classes $\{C_0, C_1, ...C_h\}$, 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 a 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.

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.

4.1.2 Offline Greedy Algorithm

First, to reduce 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). The reservation action only occurs at the beginning of an interval.

We propose a forward offline greedy algorithm (Algorithm 1) to optimize instance reservation. The basic assumption is that at the beginning of each time interval, we can look ahead at the instance demand history (matrix U), and calculate, row by row, the jth instance's utilization level during different reservation terms (e.g., 1Y, 3Y). Based on the results, 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 this jth instance. After examining j' instances needed for the upcoming interval (week), we have a provisioning plan containing each instance's pricing class assignment. We then compare this provisioning plan with the current *inventory* and decide the amount of purchased, after adjusting active reserved instances. In EC2's case, all heavy instances are re-divided according to their expiring time. Those active 3Y-Heavy instances whose expiring time shorter than 1 year are classified as 1Y-Heavy instances inventory. The purchasing decision is set as minimizing differences between new inventory and the provisioning plan,

$$\min_{R_{ik} \geq 0} (\sum_{k=1}^{h} (S_{ik} + R_{ik} - D_{ik})^2 + (h \sum_{k=1}^{h} (S_{ik} + R_{ik} - D_{ik}))^2), \quad (1)$$

where S_{ik} is the inventory of reserved instances C_k at ith time interval, D_{ik} is its provisioning plan.

4.1.3 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 $C'_k =< \tau, F_k \cdot \tau / T_k, P_k >$. Clearly, the minimal rental cost using pricing classes C' is no more than the optimal solution using pricing class C.

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

$$\sum_{ij} \min_{k} (F_k \cdot \tau / T_k + P_k \cdot U_{ij}) \quad (2)$$

The intuition here is that with the decision making interval matching the reservation period, 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 between these small intervals.

4.2 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 (1 year and 3 years in EC2's case) and cannot be "returned", demanding a load prediction mechanism that sees rather far into the future. In comparison,

Algorithm 1 Offline greedy resource provisioning

Require: $n \times m$ utilization matrix U, where n is the total number of decision intervals (the decision interval is τ), and m is the maximum capacity. Pricing classes $\{C_0, C_1, ...C_h\}$, where $C_k = \langle T_k, F_k, P_k, MUL_k \rangle$ (each item represents its reservation term, upfront fee, hourly price and minimum utilization level).

```
ReservedInstancesList.Clear()
for i ← 1 to n do
  //Calculate requirements of reserved instances
  D_i ← {0}
  for j ← 1 to m do
    for k ← h downto 1 do
      if 1/T_k · Σ_{l=i}^{i+T_k/τ−1} U_lj ≥ MUL_k then
        if U_ij/τ ≥ MUL_k or U_ij ≥ U_{i+T_k/τ,j} then
          D_ik++
        end if
        break
      end if
    end for
  end for
  //Check inventory of reserved instances
  S_i ← {0}
  for each r ∈ ReservedInstancesList do
    if r.expire_time < i then
      ReservedInstancesList.Del(r)
      next
    end if
    for k ← 1 to h do
      if r.price == P_k and r.expire_time − i < T_k/τ then
        S_ik ← S_ik + r.num
        break
      end if
    end for
  end for
  //Make plan and update reserved instances list
  R_i ← Make plan using Equation 1 from D_i and S_i
  for k ← 1 to h do
    ReservedInstancesList.Add(price=P_k,
    expire_time=i + T_k/τ − 1, num=R_ik)
  end for
end for
```

prediction-based resource management proposed in prior study focused on estimating what will happen in the next seconds or minutes [6, 9, 21].

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 4.2.1), to be coupled with conservative instance reservation strategies (Section 4.2.2).

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 5, we evaluate our online load prediction and instance provisioning algorithms using traces from these two ends of the workload dynamics spectrum.

4.2.1 Long-term Instance Demand Prediction

We aim to predict the total number of instances needed in a future time interval, as far as any current reservation is concerned. 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 [7, 11], and is widely used in business to forecast demands for inventories [11]. 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 [25, 26].

More specifically, we extend classical Holt's double-parameter ES method [11] 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:

$$m_{k+t} = d_k + v_k t + a_k t^2/2 \tag{3}$$

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:

$$\begin{aligned} d_k &= \alpha m_k + (1-\alpha)(d_{k-1} + v_{k-1} + a_{k-1}/2) \\ v_k &= \beta(d_k - d_{k-1}) + (1-\beta)(v_{k-1} + a_{k-1}) \\ a_k &= \gamma(v_k - v_{k-1}) + (1-\gamma)a_{k-1} \end{aligned} \tag{4}$$

α, β and γ are smoothing factors, with values between 0 to 1, that determine the relative weight given to recent changes vs. historical data ("responsive" vs. "smoothing") [11]. In this work, we extend the widely-used dynamic smoothing factor estimation method proposed by Trigg and Leach [32], 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 3, replacing t with w.

4.2.2 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 following the 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 (3 years in EC2's case), using the triple ES algorithm described in Section 4.2.1.
- Step 3: From the projected long-term weekly instance demand (Step 2), remove instance usage that can benefit from the current reserved instance inventory (Step 1), starting from the heaviest discount class and moving to lighter ones in a greedy manner.
- Step 4: Apply the offline instance provisioning optimization algorithm discussed in Section 4.1 on the adjusted projected weekly demand. This calculates the requirement for new reservations.
- Step 5: Make reservations according to the result of Step 4, 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. Note that we are more concerned with over-estimated future demands, which lead to overly aggressive instance reservation. By removing the "active discounted usage" allowed by the current inventory (Step 3), it reduces the impact of over-stocking. In addition, 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.

5. EVALUATION RESULTS

5.1 Experiment Setup

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 [1]. The job entries are traditional HPC jobs, usually tightly-coupled parallel applications (such as MPI programs). They are rigid jobs, each running on a fixed number of processors (nodes) during its entire execution. 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, 1640-processor IBM 655/690 system at San Diego Supercomputer Center (SDSC). 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 [17] 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 [4, 31]. For Twitter, the Google search traffic linearly correlates to its active user population (with R-square of 0.9792). Unfortunately we do not have active user population data from other SNS sites, as such data are usually considered trade secrets. We plan to carry out more verification if more user population history data become available. 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 4 and 8 years.

Cloud Platform We use Amazon's EC2 Cluster Compute Instances (CCIs), in evaluating our SEC scheduling and resource provisioning (through simulation) and our prototype SEC cluster management framework implementation (through real cloud experiments). 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 3370 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).

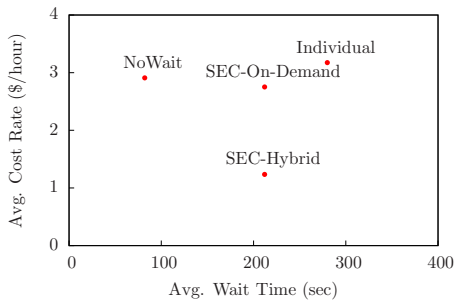
Metrics We evaluate our system with three metrics, include that average job wait time, overall system utilization, and monetary cost. The first two are standard metrics for schedulers, while the third is calculated as the overall cloud instance rental cost for SEC clusters. When using reserved instances, total cost 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 fairly, we calculate a pure cost that deduct inventory values in accordance with remaining expire period.

5.2 Cost-Responsiveness Analysis

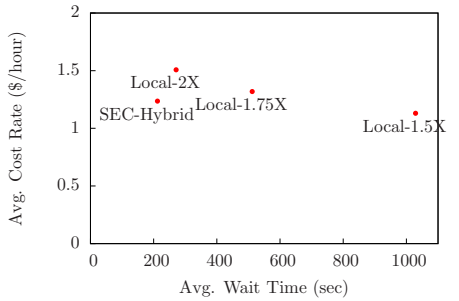
Intuitively, with workload aggregation, SEC lowers cost but incurs higher wait time. However, on-demand cloud cluster provisioning is not latency-free, as we have shown with the multi-minute instance boot delay. SEC enables instance reuse and can control the amount of wait time through the $T_{waitlmt}$ 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 request on-demand resources whenever an arriving job does not have enough nodes to start, practically eliminating the job queue, 3) two variations of SEC, with *SEC-On-Demand* using only on-demand instances and *SEC-Hybrid* using all reserved instances classes and on-demand instances. For each mode, we replayed the Data Star trace and calculated wait time and cost rate.

Figure 3a 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% and *SEC-Hybrid* obtains a 61.0% cost saving. Meanwhile, both SEC modes actually *reduces* the average job wait time of *Individual* by 24.2%,



(a) SEC vs. individual on-demand cloud execution



(b) SEC vs. owning local clusters

Figure 3: Cost-responsiveness evaluation

through instance reuse. *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*.

Table 3: In-house cluster expense items

Expense item	**Price**	**#**
Computing Nodes (DELL PowerEdge R720)	$7749	93
InfiniBand NIC	$612	93
InfiniBand Switch (36 ports)	$9172	3
I/O Nodes (DELL PowerEdge R720)	$7749	5
Disk Array (HP D2600 with 24 TB disks)	$6399	10
Fiber NIC for I/O	$1043	98
Fiber Channel Switch (16 ports)	$6599	8
Hosting for one year (energy included)	$15251	3
Total Cost (3 years)	$1108583	

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 1480 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 [36], amortizing the hardware/hosting cost over a 3 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.

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 [36]. Figure 3b shows the results, where the *Local-x*× clusters clearly demonstrate the trade-off 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.

5.3 SEC Scheduling Parameters Analysis

Our proposed job scheduling and resource provisioning algorithms in Section 3 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 decide allocated virtual machines to running jobs. In this section, we evaluate the influence of these parameters/polices. To isolate such influence, our experiments here use only on-demand instances, which leads to hourly charge rates higher than typical SEC results shown in other charts.

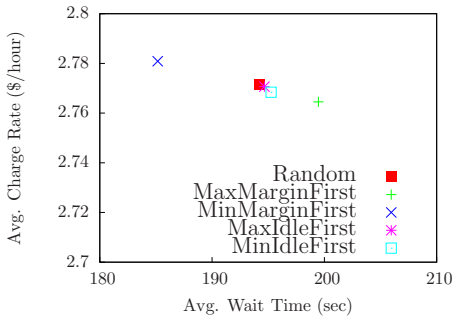
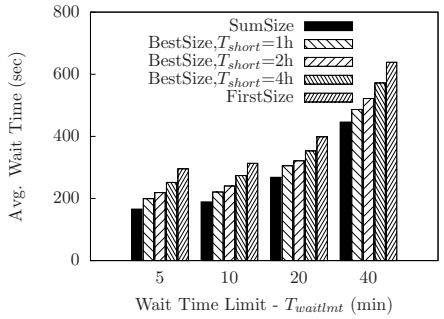


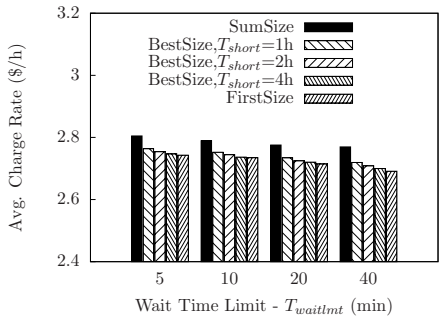
Figure 4: Impact of job placement policies

We first analyze the impact of job placement policies. Figure 4 shows simulation results using the Data Star trace, with the other two scheduling parameters fixed ($T_{waitlmt} = 5$ min, $T_{short} = 1$ hour). They indicate that *MaxMarginFirst* offers lowest cost, while *MinMarginFirst* excels at cutting

the average job wait time. The other three policies (including *Random*) perform very close to each other and provide a compromise between the two requirements.



(a) Average wait time



(b) Average charge rate

Figure 5: Impact of scheduling and provisioning parameters

Next, we analyze the impact of wait time threshold and short job threshold. Figure 5 presents simulation results using the same job trace, with job placement policy set to *MaxMarginFirst*. It shows the average wait time with different parameter configurations. Figure 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.

Figure 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 reducing wasted resources introduced by overestimated job execution time.

5.4 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.

We analyze resource usage level in classical HPC systems.

Table 4: Variance in node-hour per active user

Name	1 Day	1 Week	1 Month
SDSC Blue	0.5987	0.2434	0.1545
SDSC DataStar	0.7741	0.2533	0.1323
Sandia Ross	0.8829	0.3757	0.1933
HPC2N	1.1465	0.4969	0.3786
CTC SP2	0.4667	0.2023	0.1076
SDSC SP2	0.7468	0.2639	0.1538

As seen in Table 4, the load level per user has large fluctuations in short-term (e.g. a day/week), especially in the HPC2N trace. Meanwhile, the load level is rather stable in long term (e.g. a month), prompting a time-series method for noise removal.

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 [17]. 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. Figure 6 shows the search traffic for the six SNSs. 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).

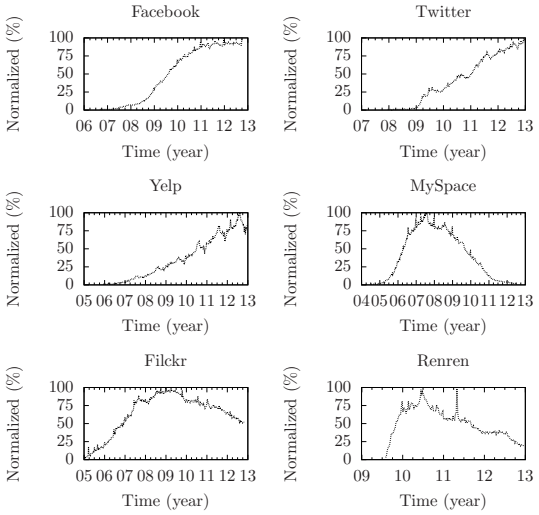
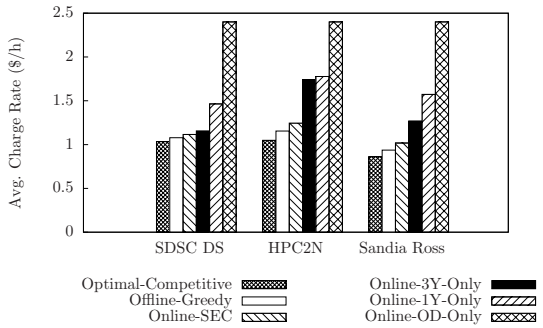
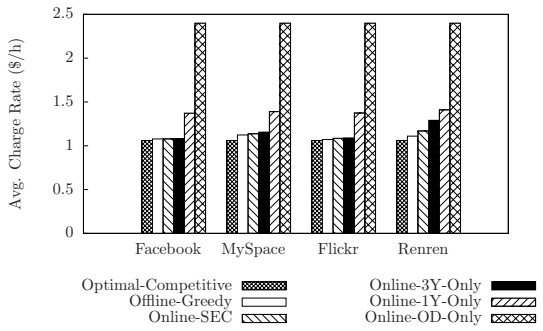


Figure 6: SNS search traffic from Google Trends

In addition to the methods proposed in Section 4, 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



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



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

Figure 7: Average charge rate using different algorithms

instance classes: on-demand (*Online-OD-Only*), 1-year reserved (*Online-1Y-Only*), and 3-year reserved (*Online-3Y-Only*).

Figure 7a shows simulation results with HPC traces. We have analyzed 6 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. Figure 7a shows that the performance of our proposed online method is very close to both offline methods. Moreover, it also shows that our method is better than the methods with only a single type of instance. Figure 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.

5.5 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 implement a proof-of-concept SEC management framework based on SLURM [2], an open-source resource manager designed for Linux clusters. We then test job submission, execution, and instance management on real Amazon EC2 resources.

Based on our results, we found the SEC overhead to be quite reasonable (at the seconds level). The major sources of overhead include protecting the per-user private temporal data, configuring newly acquired nodes, and releasing redundant nodes from queuing systems.

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 experimented with reformatting EC2 ephemeral disks and found that file system type has a dominant impact on this overhead. E.g., each EC2 845GB ephemeral disk takes, on average, 185 seconds to format with ext3, but only 3.25 seconds with ext4. With 4 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 5 below gives the average measurement. Note that releasing instances, from both EC2 and SLURM, takes around 5 seconds per instance. However, such time is not billed and not visible to users.

Table 5: 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

Finally, we again performed job scheduling simulation using the DataStar trace with varying $T_{waitlmt}$ to check the impact of the aforementioned cluster management overhead. Due to the space limit, we summarize the results rather than showing the detailed results. As expected, the second-level overhead has negligible effect on the average job wait time (0.1-2.4%).

6. RELATED WORK

Job Scheduling on Traditional HPC Systems Job scheduling strategies on traditional fixed-capacity HPC systems have been well studied. Although a series of elegant algorithms have been proposed [24,28], FCFS-based backfilling algorithm is thought to be most effective in most of cases.

There have been several mature scheduling systems being used in large HPC centers, such as LSF [18], Moab [15], Maui [19], PBS/Torque [16] and LoadLeveler [30]. In SEC system, we adopt the priority-based backfilling algorithm as our basic scheduling strategy. At the same time, the adjustment of dynamic resource scaling is also considered in our scheduling algorithm which is the main difference from traditional scheduling algorithms.

Virtual Clusters Constructing virtual clusters with grid-based virtual machines has been proposed by Figueiredo et al. [10] and researchers also implemented such systems, such as In-VIGO [27] and VMPlants [22]. Although they share the on-demand nature with cloud-based clusters, grid virtual clusters were designed to use distributed parallel platforms (mostly for running throughput-oriented applications). The lack of commercial production support also prevents such virtual systems to become practical alternatives to organization-owned physical clusters.

Recently, some projects are focusing on facilitating cloud computing [8, 23, 33]. 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.

Several scheduling algorithms for cloud-based virtual clusters have been proposed. Moschakis et al. explored the usage of two common Gang Scheduling strategies [29]. Genaud et al. evaluated the strategies based on classic scheduling and bin-packing algorithms with Amazon's pricing model [12]. Although these studies consider the trade-off between cost and response time, they have not combined batch scheduling algorithms with dynamic cloud resource scaling.

Cost-effective Cloud Computing Many researchers have presented some work about cost minimization on public cloud platform with probabilistic model [5], analytical performance price model [37] and spot instances [20, 35]. Our proposed SEC model is a beneficial complement for these work through load aggregation and instances reuse.

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. Zheng [38] and Kalyvianaki [21] 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 [9]. Ardagna et al. used a simple Exponential Smoothing to predict jobs arrival rate in cloud platform [6]. Those work mainly focus on short-term analysis while our study is a long-term prediction problem which is more complex.

7. DISCUSSION

Purchasing redundant reserved instances is risky. In SEC system, we reduce the risk by aggregating users demand and an online prediction-based method for instances provisioning. Recently, Amazon developed a reserved instances marketplace [13], to help users to trade their idle resources. It can reduce the risk of operating a SEC system further.

The type of spot instance is another pricing model introduced by Amazon, which can significantly lower rental cost for time-flexible, interruption-tolerant tasks [14]. The price of spot instances is often significantly cheaper than on-demand instances for the same EC2 instance types. In SEC system, we can use spot instances to process short jobs to further reduce rental costs. We leave this solution as our future work.

8. 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. From our design and evaluation, we argue that SEC can potentially become a viable alternative to organizations owning and managing physical clusters. In addition, we have found that by performing load aggregation and instance reuse simultaneously, SEC can deliver significantly better cost-effectiveness without hurting user experience, compared to using the standard on-demand cloud clusters.

Acknowledgments

We want to express our thanks to the anonymous reviewers, for providing valuable suggestions that have helped to improve the quality of the manuscript. In particular, we thank our shepherd, Henry Tufo, for providing guidance and feedback during our final paper preparation. This work has been supported partly by the Chinese National High-Tech Research and Development Plan (863 project) 2012AA01A302, as well as NSFC project 61232008 and 61103021. It has also been sponsored by NSF grants 0546301 (CAREER), 0915861, 0937908 and 0958311, in addition to a joint faculty appointment between ORNL and NCSU, plus a senior visiting scholarship at Tsinghua University.

9. REFERENCES

[1] Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`, 2012.

[2] SLURM: A Highly Scalable Resource Manager. `https://computing.llnl.gov/linux/slurm/`, 2012.

[3] StarCluster. `http://web.mit.edu/star/cluster/`, 2012.

[4] C. Alex and E. Mark. An In-Depth Look Inside the Twitter World. `http://blog.rjmetrics.com/new-data-on-twitters-users-and-engagement/`, 2009.

[5] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 257–266. IEEE, 2010.

[6] D. Ardagna, S. Casolari, and B. Panicucci. Flexible distributed capacity allocation and load redirect algorithms for cloud systems. In *Cloud Computing (CLOUD)*, pages 163–170. IEEE, 2011.

[7] B. Billah, M. L. King, R. D. Snyder, and A. B. Koehler. Exponential smoothing model selection for forecasting. *International Journal of Forecasting*, 22(2):239–247, 2006.

[8] F. Doelitzscher, M. Held, C. Reich, and A. Sulistio. Viteraas: Virtual cluster as a service. In *Cloud*

Computing Technology and Science (CloudCom). IEEE, 2011.

[9] T. V. T. Duy, Y. Sato, and Y. Inoguchi. Performance evaluation of a green scheduling algorithm for energy savings in cloud computing. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[10] R. Figueiredo, P. Dinda, and J. Fortes. A case for grid computing on virtual machines. In *Distributed Computing Systems*. IEEE, 2003.

[11] E. S. Gardner Jr. Exponential smoothing: The state of the art. *Journal of Forecasting*, 4(1):1–28, 1985.

[12] S. Genaud and J. Gossa. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *Cloud Computing (CLOUD)*. IEEE, 2011.

[13] A. Inc. Amazon EC2 Reserved Instance Marketplace. `http://aws.amazon.com/ec2/reserved-instances/marketplace/`, 2012.

[14] A. Inc. Amazon EC2 Spot Instances. `http://aws.amazon.com/ec2/spot-instances/`, 2012.

[15] A. C. E. Inc. MOAB workload manager. `http://www.supercluster.org/moab/`, 2012.

[16] A. C. E. Inc. PBS/Torque user manual. `http://www.clusterresources.com/torquedocs21/usersmanual.shtml`, 2012.

[17] G. Inc. Google Trends. `http://www.google.com/trends/`, 2013.

[18] P. C. Inc. Platform LSF. `http://www.platform.com/products/LSFfamily/`, 2012.

[19] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing*. Springer, 2001.

[20] B. Javadi, R. Thulasiramy, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *Utility and Cloud Computing (UCC)*. IEEE, 2011.

[21] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th international conference on Autonomic computing*, pages 117–126. ACM, 2009.

[22] I. Krsul, A. Ganguly, J. Zhang, J. Fortes, and R. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Proceedings of the ACM/IEEE SC04 Conference*, 2004.

[23] J. Lucas-Simarro, R. Moreno-Vozmediano, R. Montero, and I. Llorente. Scheduling strategies for optimal service deployment across multiple clouds. *Future Generation Computer Systems*, 2012.

[24] S. Majumdar, D. Eager, and R. Bunt. *Scheduling in multiprogrammed parallel systems*, volume 16. ACM, 1988.

[25] 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. *Journal of forecasting*, 1(2):111–153, 1982.

[26] S. Makridakis and M. Hibon. The m3-competition: results, conclusions and implications. *International Journal of Forecasting*, 16(4):451–476, 2000.

[27] A. Matsunaga, M. Tsugawa, S. Adabala, R. Figueiredo, H. Lam, and J. Fortes. Science gateways made easy: the in-vigo approach. *Concurrency and Computation: Practice and Experience*, 19(6), 2007.

[28] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for iviukiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2), 1993.

[29] I. Moschakis and H. Karatza. Evaluation of gang scheduling performance and cost in a cloud computing system. *The Journal of Supercomputing*, 2011.

[30] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY – LoadLeveler API project. In *Job Scheduling Strategies for Parallel Processing*. Springer, 1996.

[31] J. Stein. New Data on Twitter's Users and Engagement. `http://blog.rjmetrics.com/new-data-on-twitters-users-and-engagement/`, 2010.

[32] D. Trigg and A. Leach. Exponential smoothing with an adaptive response rate. *OR*, pages 53–59, 1967.

[33] W. Voorsluys, S. Garg, and R. Buyya. Provisioning spot market cloud resources to create cost-effective virtual clusters. *Algorithms and Architectures for Parallel Processing*, 2011.

[34] E. Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. *Login*, 33(5), 2008.

[35] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *Cloud Computing (CLOUD)*. IEEE, 2010.

[36] 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 *State of the Practice Reports*, page 11. ACM, 2011.

[37] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang. Optimal resource rental planning for elastic applications in cloud market. In *Parallel & Distributed Processing Symposium (IPDPS)*, pages 808–819. IEEE, 2012.

[38] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai. Tracking time-varying parameters in software systems with extended kalman filters. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 334–345. IBM Press, 2005.