Accurate Resource Prediction for Hybrid IaaS Clouds Using Workload-Tailored Elastic Compute Units

Shigeru Imai, Thomas Chestna, Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
110 Eighth Street, Troy, NY 12180, USA
Email: {imais,chestt2,cvarela}@cs.rpi.edu

***Abstract*—Cloud computing's pay-per-use model greatly reduces upfront cost and also enables on-demand scalability as service demand grows or shrinks. Hybrid clouds are an attractive option in terms of cost benefit; however, without proper elastic resource management, computational resources could be over-provisioned or under-provisioned, resulting in wasting money or failing to satisfy service demand. In this paper, to accomplish accurate performance prediction and cost-optimal resource management for hybrid clouds, we introduce *Workload-tailored Elastic Compute Units (WECU)* as a measure of computing resources analogous to Amazon EC2's ECUs, but customized for a specific workload. We present a dynamic programming-based scheduling algorithm to select a combination of private and public resources which satisfy a desired throughput. Using a loosely-coupled benchmark, we confirmed WECUs have 24% better runtime prediction ability than ECUs on average. Moreover, simulation results with a real workload distribution of web service requests show that our WECU-based algorithm reduces costs by 8-31% compared to a fixed provisioning approach.**

I. Introduction

Growing demand for "Big Data" analytics applications requires a corresponding growth in "big computing" resources, making hybrid clouds an increasingly attractive infrastructure. A hybrid cloud enables users who have access to their own private cloud to scale out to public computing resources only occasionally. Especially for massively-parallel applications, for which the "cost-associativity" quality of the cloud computing model holds [1], we can get results faster by paying more money for public cloud resources; however, to orchestrate distributed virtual machine (VM) instances in hybrid clouds in a cost-efficient manner is a non-trivial task.

Amazon created the *Elastic Compute Unit (ECU)* as a measure of computing power of their various VM instance types depending on the price and quality of service (QoS) they offer to help users choose appropriate virtual environments. Their explanation of ECU is as follows: "We use several benchmarks and tests to manage the consistency and predictability of the performance of an EC2 Compute Unit. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor" [2]. ECU helps as a relative performance measure for a wide range of workloads, but it cannot be accurate for all types of workloads. Moreover, in the hybrid cloud model, ECU is not defined for private computing resources. Therefore, to accurately schedule resources not only on the public cloud, but also on the private cloud, the computational power of private computing resources must be quantified for better hybrid cloud performance prediction.

To get accurate performance predictions and associated cost reduction over hybrid clouds, we introduce the notion of *Workload-tailored Elastic Compute Unit (WECU)* as a unit of computing power for a specific type of workload on a specific virtual or physical machine type. WECU is obtained by actually running the target workloads on available VM instances on the target machine for a short period. Thus, the use of WECU is only meaningful for long running workloads such as computationally-intensive scientific workloads and web services. In this paper, we choose farmer-worker loosely-coupled computationally intensive workloads and a web service receiving fluctuating requests as benchmarks of long-running workloads. With these benchmarks, we compare the performance and cost prediction capability of using WECUs vs. using Amazon EC2's ECUs.

In addition to improving the performance prediction based on WECU, we also consider application-level migration to support dynamic workload scalability. When dynamically scaling up the computation of long-running farmer-worker programs, migrating the workers from a private cloud to a public cloud is an effective approach because it is light-weight compared to VM-level migration and is able to let the workers continuously run without restarting [3] . The authors previously worked on application-level migration within private clouds using the actor-oriented programming language called SALSA (Simple Actor Language System and Architecture) [4], which supports transparent application-level migration at the language level. In this paper, we implement the farmer-worker programs in SALSA and simulate fluctuating web service scalability based on application-level migration.

In summary, the main contributions of this paper are as follows. First, it presents the notion of WECU and shows its performance predictability together with a cost-optimal throughput-constrained resource provisioning algorithm. Sec-

ond, it presents a simulation of dynamic workload scaling using a real web service request workload distribution. WECUs have significantly better performance predictability which results in significant improvements in QoS over ECU-based resource management, and significant cost savings compared to over-provisioning approaches.

The rest of the paper is organized as follows. Section II describes technical background. Section III introduces the concept of Workload-tailored ECU with preliminary results for a loosely-coupled benchmark. Section IV describes the throughput-constrained resource configuration algorithm for hybrid clouds based on dynamic programming. Section V describes the hybrid cloud management middleware applying the algorithm mentioned in Section IV. Section VI presents the results of performance prediction in a real hybrid cloud as well as simulation results of dynamic scaling of web request workloads. Section VII describes related work, and finally, Section VIII concludes the paper and describes potential future directions.

II. Technical Background

A. SALSA Programming Language

SALSA is a general-purpose actor-oriented programming language. It is especially designed to facilitate the development of dynamically reconfigurable distributed applications as needed for example for truly elastic cloud computing. SALSA supports the actor model's unbounded concurrency, asynchronous message passing, and state encapsulation [5], [6]. In addition to these features, it implements a universal naming model for application semantics-preserving actor migration and location-transparent message sending.

B. Workload Scalability over a Hybrid Cloud

We use application-level migration as a means to support dynamic workloads. When dynamically scaling up the computation of farmer-worker programs, we can migrate some of the workers from one node to another which gives them extra computing power. SALSA actors do not leave any residual dependencies [7] or create resource conflicts when moving around computing nodes. This approach is effective because it is light-weight compared to other migration techniques such as VM-level migration [8] and is able to keep the workers running without restarting the application as opposed to check-point restart approaches [9].

Moreover, we can not only scale up and down the computation within a private cloud by migrating actors, but also scale it out to a public cloud transparently from the application's point of view (See Figure 1).

III. Workload-tailored ECU

There are quite a few VM instance types available in Amazon EC2 associated with various ECUs and prices. Some of the instance types are shown in Table I. To find out the actual performance difference between instance types, we did a preliminary experiment evaluating a simple benchmark application called `Trap` on each instance type as well as on

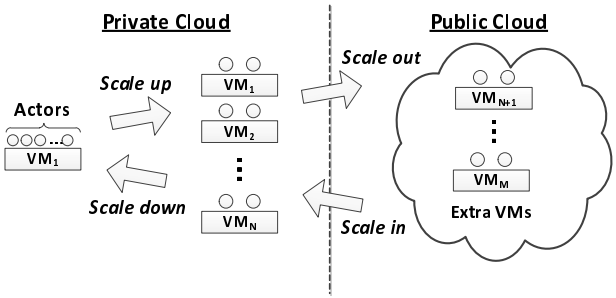

Fig. 1. Workload scalability over a hybrid cloud (taken from [3]).

two other nodes, A and B, in our private cloud. `Trap` is a massively parallel application written in SALSA and computes a trapezoidal numerical integration for a given function with a given interval (see Section VI for details). The task size is defined as the number of trapezoids.

TABLE I
Amazon EC2 On-Demand Instances (as of September 2013).

Instance Type	vCPU	ECU	Price[USD]	Price/ECU
m1.small	1	1	0.06	0.06
m1.medium	1	2	0.12	0.06
m1.large	2	4	0.24	0.06
m1.xlarge	4	8	0.48	0.06
m3.xlarge	4	13	0.5	0.038
m3.2xlarge	8	26	1	0.038
c1.medium	2	5	0.145	0.029
c1.xlarge	8	20	0.58	0.029

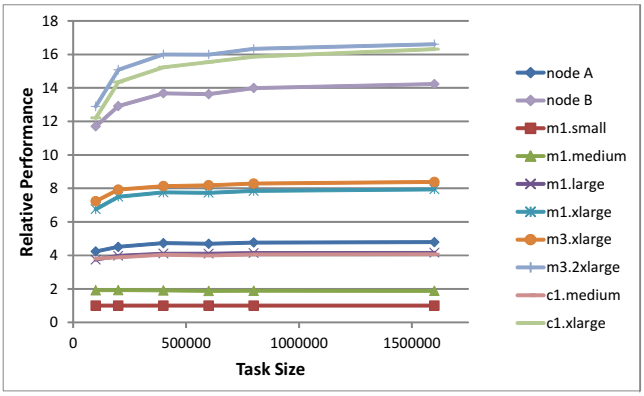

Fig. 2. Relative performance of Amazon EC2 instances and private physical nodes (node A and B). The performance of `m1.small` is 1.0.

For task sizes ranging from 100,000 to 1,600,000, we observe relative runtime performance results in Figure 2. In this figure, the base performance is defined as the runtime for the `m1.small` instance and the other relative performances are divided by the `m1.small`'s runtime. As the graph clearly shows, ECU does not always show the exact performance differences for the `Trap` application. For example, `c1.medium` has five ECUs while its relative performance is only about four, also `c1.xlarge` has 26 ECUs while its relative performance is ranging from 12 to a little over 16.

TABLE II
ECU AND WECU VALUES FOR THE TRAP APPLICATION (1 WECU = 2758.54 [TASKS/SEC]).

Instance Type	ECU	WECU	Price/WECU
m1.small	1	1.0	0.06
m1.medium	2	1.882	0.0632
m1.large	4	4.156	0.0594
m1.xlarge	8	7.940	0.0617
m3.xlarge	13	8.380	0.0610
m3.2xlarge	26	16.604	0.0615
c1.medium	5	4.070	0.0359
c1.xlarge	20	16.316	0.0366
node A	n/a	4.795	n/a
node B	n/a	14.223	n/a

These results tell us that ECUs are not very accurate in predicting the performance of the `Trap` application, and thus we need to define a new performance measurement unit called *Workload-tailored Elastic Compute Unit (WECU)*. WECU is defined as follows: *one WECU corresponds to the throughput of a one-ECU instance on a specific workload.* General WECU values associated with arbitrary instance types processing the same workload are defined by dividing their throughput by the base throughput. In short, it is a relative performance unit based on the actual workload throughput. For the `Trap` application and instances used in the preliminary experiment, we define the base throughput as `m1.small`'s throughput, that is, 1 WECU = 2758.54 [tasks/sec]. All obtained values of WECUs are shown in Table II. Please note these values are computed after the performance converges. Since we will use WECU for long-running workloads, we want to use the throughput in steady-state conditions.

IV. RESOURCE CONFIGURATION ALGORITHM

The resource configuration algorithm consists of four steps as shown below to support cost-optimal throughput-constrained resource management.

- **Step 1.** Compute target throughput and set a corresponding computational power requirement.
- **Step 2.** Check if the private cloud can satisfy the computational power requirement.
- **Step 3.** If Step 2 cannot satisfy the requirement, use extra resources in the public cloud.
- **Step 4.** Assign workers to the instances allocated in Steps 2 and 3.

The details of each step are explained below in order.

Step 1: Target Throughput and Computational Requirement

Given variables:

- $t_{deadline}$: deadline to finish the tasks.
- t_{curr}: current time.
- $tasks$: total number of tasks to process.
- $tasks_{done}$: total number of completed tasks.
- λ: throughput per WECU.
- Δ_{curr}: current throughput.
- τ: throughput threshold for activating reconfiguration.

The target throughput Δ_{target} is computed as follows[1]:

$$\Delta_{target} = (tasks - tasks_{done})/(t_{deadline} - t_{curr}).$$

If $|\frac{\Delta_{target}-\Delta_{curr}}{\Delta_{target}}| < \tau$, the algorithm does nothing and quits, otherwise it computes a required computational power η_{target} as follows:

$$\eta_{target} = \Delta_{target}/\lambda. \quad (1)$$

Note that η_{target} is in WECU units, which is defined in Section III. We need to achieve η_{target} with computing instances collectively allocated from the private and public clouds so that the target throughput Δ_{target} can be maintained.

Step 2: Private Cloud Resource Configuration

Given variables:

- $R_{priv} = \{(type_1, \eta_1, cpu_1, num_1), ..., (type_N, \eta_N, cpu_N, num_N)\}$: N types of VM instances available in the private cloud, where $type_i$ is the VM instance type, η_i is the computational power, cpu_i is the number of virtual CPUs, and num_i is the available number of the ith instance type respectively.
- η_{target}: target computational power computed in Equation 1.

Algorithm 1 outputs the following:

- A_{priv} : a set of instances to be allocated in the private cloud. ith element is a 4-tuple $(type_i, \eta_i, cpu_i, num_i)$.
- σ_{priv} : total computational power provided by A_{priv}.
- η_{remain} : remaining computational power needed to satisfy target throughput.

input : R_{priv}, η_{target}
output: $A_{priv}, \sigma_{priv}, \eta_{remain}$
$A_{priv} = \emptyset$; $\sigma_{priv} = 0.0$; $\eta_{remain} = \eta_{target}$; $i = 1$;
while $i \leq N$ and $0 < \eta_{remain}$ **do**
  if $\lceil \eta_{remain}/\eta_i \rceil \leq num_i$ **then**
    $num = \lceil \eta_{remain}/\eta_i \rceil$;
  end
  else
    $num = num_i$;
  end
  $A_{priv} = A_{priv} \cup \{(type_i, \eta_i, cpu_i, num)\}$;
  $\sigma_{priv} = \sigma_{priv} + num \times \eta_i$;
  $\eta_{remain} = \eta_{remain} - num \times \eta_i$;
  $i = i + 1$;
end
return A_{priv}, η_{remain};

Algorithm 1: Private Cloud Resource Configuration.

Algorithm 1 allocates resources giving priority to instance types in the order they appear in R_{priv}. It simply deducts available computing power from η_{remain}. If $\eta_{remain} \leq 0$,

[1]For non-terminating workloads, we assume a target throughput is given in tasks/second.

then the algorithm outputs A_{priv} and goes to Step 4 to assign workers, otherwise it proceeds to Step 3 to further allocate more instances from the public cloud.

Step 3: Cost Optimal Public Cloud Resource Configuration

Given variables:

- $R_{pub} = \{(type_1, \eta_1, cpu_1, price_1), ..., (type_M, \eta_M, cpu_M, price_M)\}$: M types of VM instances available in the public cloud, where $type_i$ is the VM instance type, η_i is the computational power, cpu_i is the number of virtual CPUs, and $price_i$ is the hourly price of the ith instance type respectively.
- η_{target}: η_{remain} obtained in Step 2.

The minimal cost that satisfies arbitrary computational power η is given by recursive Equation 2. It compares the cost of choosing instance i out of M instance types recursively until it finds an instance which has larger computational power than the required η. Using a dynamic programming algorithm directly derived from this equation, the following variables are obtained:

- A_{pub}: instances to be allocated from the public cloud
- $cost_{pub}$: total cost of A_{pub}
- σ_{pub}: total computational power of the instances in A_{pub},

where

$$\begin{aligned} A_{pub} &= \{(type_1, \eta_1, price_1, num_1), ..., \\ &\quad (type_L, \eta_L, price_L, num_L)\}, \\ cost_{pub} &= \sum_{i\text{th instance type } \in A_{pub}} price_i \times num_i, \\ \sigma_{pub} &= \sum_{i\text{th instance type } \in A_{pub}} \eta_i \times num_i. \end{aligned}$$

$$COST(\eta) = \min_{1 \le i \le M} \begin{cases} price_i & (\eta \le \eta_i) \\ price_i + & \text{(otherwise)} \\ COST(\eta - \eta_i) \end{cases} \quad (2)$$

Our dynamic programming algorithm's running time is $O(\eta M)$. Cost-optimality of Equation 2 can be visually confirmed by Figure 3. As the figure shows, obtained resource configurations produce cost-optimal configurations among other possible ones.

Step 4: Workers assignment

Given $tasks, A_{priv}, \sigma_{priv}, A_{pub}$ and σ_{pub} from the previous steps, Algorithm 2 outputs the following worker assignment:

- W_{priv}: contains a set of tuples $(worker_i, tasks_i)$ for instance type i, where $worker_i$ is the number of workers and $tasks_i$ is the number of tasks to be assigned to instances of type i.
- W_{pub}: contains the same information as above for the public cloud.
- $tasks_{per\_worker}$: the number of tasks per worker.

Algorithm 2 assigns tasks in proportion to an instance type's computational power while keeping the number of tasks for each worker constant. By assigning tasks to workers this way, we are able to balance the load between heterogeneous VM instances just by migrating workers. Also, $tasks_{per\_worker}$ is determined in connection to the number of virtual CPUs. This is because it is known that the granularity of workers has an impact on the workload performance on a given number of processors [10]. To be in a region of high performance, $tasks_{per\_worker}$ is chosen so that the number of workers is at least twice as many as the number of virtual CPUs.

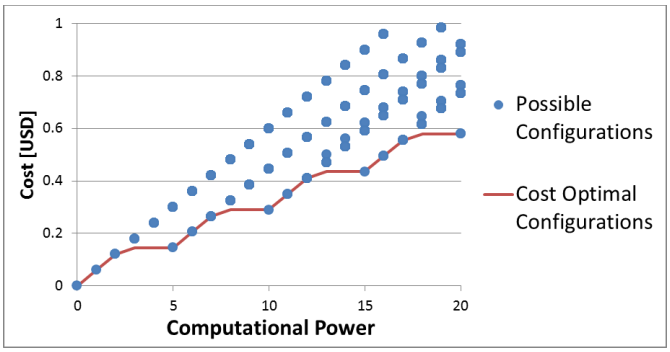

Fig. 3. Cost-optimal configurations among other possible configurations.

input : $tasks, A_{priv}, \sigma_{priv}, A_{pub}, \sigma_{pub}$
output: $W_{priv}, W_{pub}, tasks_{per\_worker}$
$W_{priv} = \emptyset$; $W_{pub} = \emptyset$;
foreach instance type i in A_{priv} **do**
  $tasks_i = tasks \times \frac{\eta_i}{\sigma_{priv}+\sigma_{pub}} \times \frac{1}{num_i}$;
end
foreach instance type j in A_{pub} **do**
  $tasks_j = tasks \times \frac{\eta_j}{\sigma_{priv}+\sigma_{pub}} \times \frac{1}{num_j}$;
end
$tasks_{per\_worker} = \min_{1 \le i \le N, 1 \le j \le M} \{\frac{tasks_i}{cpu_i \times 2}, \frac{tasks_j}{cpu_j \times 2}\}$;
foreach instance type i in A_{priv} **do**
  $worker_i = \lceil \frac{tasks_i}{tasks_{per\_worker}} \rceil$;
  $W_{priv} = W_{priv} \cup \{(worker_i, tasks_i)\}$;
end
foreach instance type j in A_{pub} **do**
  $worker_j = \lceil \frac{tasks_j}{tasks_{per\_worker}} \rceil$;
  $W_{pub} = W_{pub} \cup \{(worker_j, tasks_j)\}$;
end
return $W_{priv}, W_{pub}, tasks_{per\_worker}$;

Algorithm 2: Workers Assignment.

V. Hybrid Cloud Management Middleware

The system architecture of the middleware that implements the resource management algorithm for the hybrid cloud is shown in Figure 4.

This middleware interacts with the user and the hybrid cloud as follows:

1) **Submit a config file**: The user prepares a configuration file that contains deadline, application to execute, num-

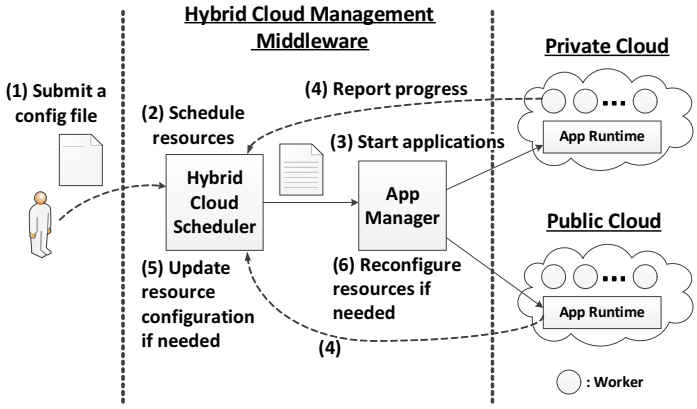

Fig. 4. System Architecture of the Hybrid Cloud Management Middleware.

ber of tasks, throughput per one WECU, and information about clouds which the user wants to use (*i.e.*, credential, cost, and WECU for each instance type, and so on), and then the user submits the configuration file to the Hybrid Cloud Scheduler.

2) **Schedule resources**: Based on the algorithm described in Section IV, the Hybrid Cloud Scheduler computes required resources to achieve the throughput derived from the given deadline. The generated resource configurations (*i.e.*, $A_{priv}, A_{pub}, W_{priv}, W_{pub}, tasks_{per\_worker}$) are passed to the App Manager.
3) **Start applications**:The App Manager accordingly starts application runtimes such as Xen and Java VMs. Subsequently, it distributes workers over the private and public clouds to start the application.
4) **Report progress**: The workers constantly report their progress to the Hybrid Cloud Scheduler.
5) **Update resource configuration**: The Hybrid Cloud Scheduler periodically executes Step 1 of the algorithm presented in Section IV. If the current throughput Δ_{curr} diverges from the target throughput Δ_{target} by more than threshold τ, it executes the rest of the steps and schedules the new resource configurations to occur right before the next billing period.
6) **Reconfigure resources**: Once the instance allocations are done, the App Manager, instead of reflecting newly obtained worker assignments (*i.e.*, $W_{priv}, W_{pub}, tasks_{per\_worker}$), migrates the existing workers to balance the load in the new virtual configuration.

VI. Evaluation

In this section, the runtime predictability, consequent QoS, and cost saving ability of the WECU-based resource configurations are evaluated and compared against ECU-based approaches.

A. Workload: Trapezoidal Numerical Integration

The `Trap` application written in SALSA computes an approximated value of $\int_a^b f(x)dx$. The farmer actor breaks the interval $[a, b]$ into n trapezoids and assigns a certain number of trapezoids to each actor program. After all the worker actors compute the value of f for given trapezoids, the farmer sums up all the partial trapezoid values returned from workers. Since the computation of each trapezoid does not depend on others, each worker can work without communicating with other workers.

B. Experimental Settings

Hybrid Cloud Environment: We have two host machines, node A and B, as the private cloud hosts. Node A has AMD Opteron 848 processor (4 cores) running at 2.2 GHz and 15 Gbytes of memory. Node B has Intel Xeon CPU E31220 processor (4 cores) running at 3.1 GHz and 6 Gbytes of memory. We dedicate node A to the name service required by SALSA and the farmer actor while node B is used to run worker actors. As the public cloud, we use Amazon EC2 with VM instance types presented in Table I. Also, in the following experiments, information shown in Table II is used as computational power in ECU and WECU units.

ECU-based approach: The resource configuration algorithm presented in Section IV is also applicable for ECU, if ECU is used as a unit of computational power η. Therefore, we use ECU values depicted in Table II and compare the same algorithm with WECU and ECU units respectively. Note that for nodes A and B, 4 and 14 are used as computational power η for the ECU-based approach.

C. Experiment 1: Runtime Prediction

The `Trap` application is tested in the hybrid cloud with the following conditions:

- $t_{deadline}$: 60 seconds.
- $tasks$ (the number of trapezoids): $\{3 \times 10^6, 6 \times 10^6, 9 \times 10^6, 12 \times 10^6\}$.
- over-provisioning rate [%]: $\{0, 2, 4, 6, 8, 10\}$.

Over-provisioning is typically used to account for the lack of accuracy in predicting workload performance. For example, if the over-provisioning rate is 10%, then the over-provisioned deadline will be 54 seconds and the algorithm takes this value as the new deadline $t_{deadline}$.

The runtimes results for (a) ECU-based and (b) WECU-based approaches are shown in Figure 5. Also, the throughput and costs observed in the same experiments are shown in Figure 6 and 7 respectively.

From Figure 5, we can clearly tell that the WECU-based resource configuration better predicts the actual performance compared to the ECU-based approach. The average differences between initial predicted runtime and the actual runtime are: 4.59% for WECU and 29.25% for ECU. Even though initial worker distribution overhead is not accounted, WECU-based approach succeeded to meet the deadline for over-provisioning rates larger than 6%. This prediction performance difference

can be explained in connection with throughput and costs: As shown in Figure 7, the ECU-based approach spends less money than the WECU-based approach, resulting in less throughput as shown in Figure 6. In other words, the ECU-based approach overestimates the performance of some of the VM instances resulting in significantly more missed deadlines.

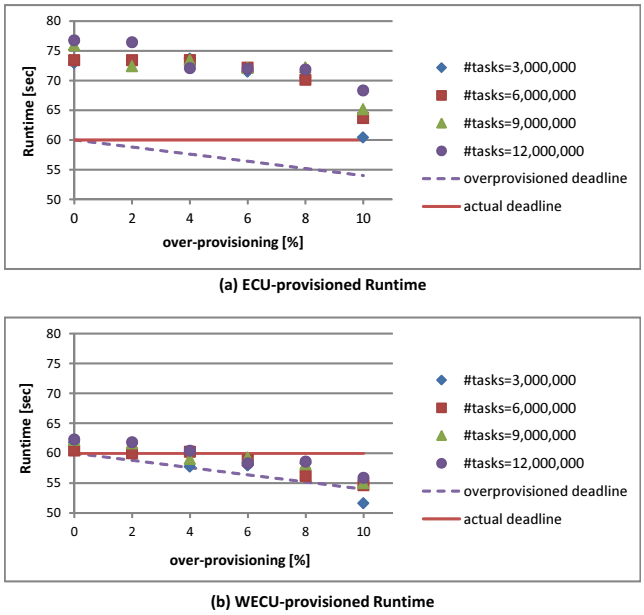

Fig. 5. Runtime results for (a) ECU-based and (b) WECU-based resource configurations.

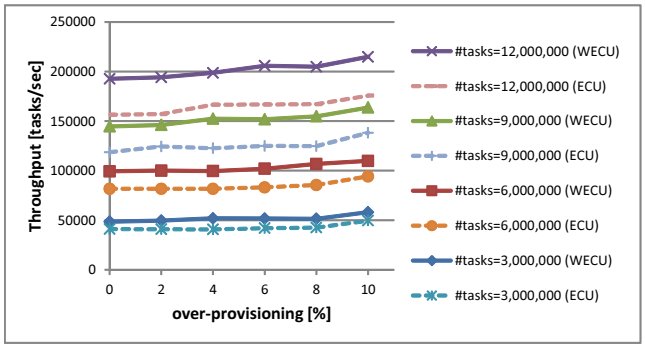

Fig. 6. Throughput results for ECU-based and WECU-based resource configurations.

D. Experiment 2: Throughput/Cost Simulation for Fluctuating Web Requests

To simulate a real workload, daily access statistics for the cloud computing article of Wikipedia [11] were used. The data is for the month of June 2013 and contains the number of daily requests to the article. The purpose of the experiment is two-fold. First, we evaluate how well both WECU and ECU-based approaches satisfy dynamically changing requests. Second, we evaluate cost savings achieved with our resource management algorithm against fixed-provisioning approaches.

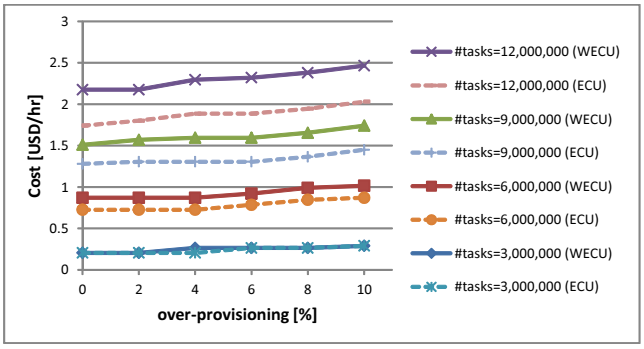

Fig. 7. Cost used for ECU-based and WECU-based resource configurations.

Hourly throughput (*i.e.*, processed requests per hour) is shown in Figure 8, which has characteristic patterns for 1) weekdays, 2) weekends, and 3) a spike observed on June 24th. We performed real hybrid cloud experiments for one hour for the said three cases each with configurations provided by WECU-based and ECU-based approaches. Also, to define the request processing time, the time to process one Wikipedia request was assumed to be equivalent to the time to process 500,000 `Trap` application tasks. With $t_{deadline}$ = 1 hour, the number of tasks used as $tasks$ inputs to the resource management algorithm are as follows. For the weekends and weekdays, we used the maximum numbers of requests during these periods excluding June 24th (Mon). For the spike of June 24th, however, due to Amazon EC2's instance creation quota (in our case, 20 instances), we could not create instances for the original 2720 requests per hour for the spike because 27 instances were required (see Table III for detail resource configurations). Thus, we reduced requests for the spike from 2720 to 1887 requests per hour to fit the Amazon quota limit for the experiment.

1) **Weekends**: 539 [requests] = 269,541,667 [tasks]
2) **Weekdays**: 773 [requests] = 386,479,167 [tasks]
3) **Spike (original)**: 2720 [requests] = 1,360,437,500 [tasks]
4) **Spike (reduced)**: 1887 [requests] = 943,770,834 [tasks]

We chose 6% over-provisioning rate for this experiment based on results from Experiment 1. It is expected for the WECU-based approach to meet the throughput constraint with this over-provisioning rate. So, we set the baseline where the WECU-based approach would perform reasonably well and quantified how many requests the ECU-based approach would miss.

From the parameters mentioned above, we got both WECU-based and ECU-based resource configurations and associated costs as shown in Table III. Using these configurations, we conducted experiments with the `Trap` application and obtained the throughput results in Figure 8.

As expected, the WECU-based approach satisfies all the requests including the reduced spike on June 24th, whereas the ECU-based approach fails to satisfy the demand 11 days of the month. For the ECU-based approach, the average number

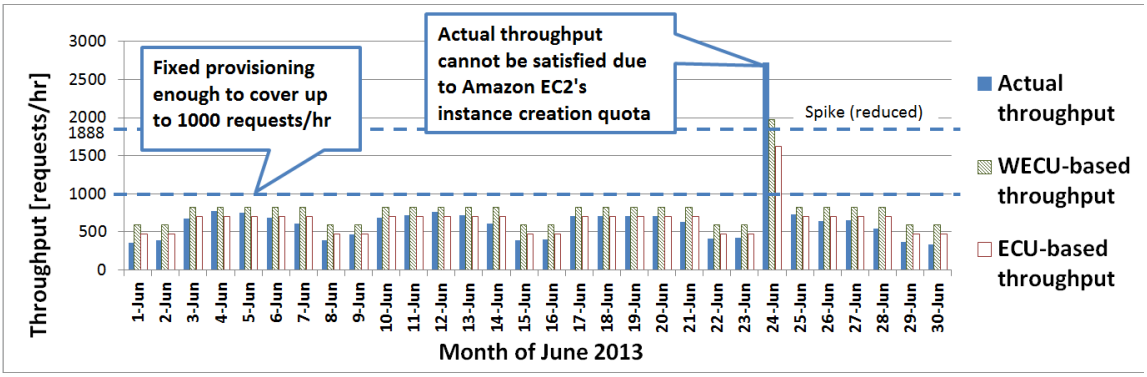

Fig. 8. Throughput simulation results for workload distribution based on Wikipedia 'cloud computing' article access statistics.

TABLE III
RESOURCE CONFIGURATION AND ASSOCIATED COSTS FOR WORKLOAD DISTRIBUTION BASED ON WIKIPEDIA 'CLOUD COMPUTING' ARTICLE ACCESS STATISTICS.

Category	Resource Configurations		Cost [USD/hr]	
	WECU	ECU	WECU	ECU
Weekends	node B: 1 c1.medium: 4	node B: 1 c1.medium: 3 m1.small: 1	0.58	0.495
Weekdays	node B: 1 c1.medium: 3 c1.xlarge: 1	node B: 1 c1.medium: 2 c1.xlarge: 1	1.015	0.87
Spike (original)	node B: 1 c1.medium: 25 c1.xlarge: 2	node B: 1 c1.medium: 19 c1.xlarge: 2	4.785	3.915
Spike (reduced)	node B: 1 c1.medium: 14 c1.xlarge: 2	node B: 1 c1.medium: 10 c1.xlarge: 2	3.19	2.61

of missed requests per hour is 18.67, which is equivalent to 13,446 requests per month.

Looking at Table III, even though the WECU-based approach costs more money than the ECU-based approach, the former approach successfully generates computationally more powerful configurations than the latter does. These cost spending for the WECU-based approach is justified since it generates higher throughput, and therefore, it succeeds to satisfy required throughput demands except for the spike on June 24th.

If a cloud service does not have dynamic scalability unlike our proposed middleware, it has to statically provision the resources. If resource configurations generated with the WECU-based approach that satisfies throughput up to 1000 [requests/hr] is used, it would cost 1044 [USD/month] using Amazon EC2 on-demand instances and they could not cover the sudden spike on June 24th. If Amazon EC2 reserved instances are available under the same condition, the WECU-based approach could generate a resource configuration that cost as little as 648 [USD/month] with ten `c1.medium` instances; however, to use these instances with a discounted hourly price (0.09 [USD/hr], price as of September 2013), you need to pay additional 1610 USD (=161 * 10) as one year reservation fee. So, the reserved instance based approach costs 648 + 1610/12 = 782.16 [USD/month] on average. On the other hand, if they have a dynamic provisioning capability with the WECU-based approach, the total cost per month would be 716.88 (=24 hr * (0.58*10 days + 1.015*19 days + 4.785*1 day)) [USD/month] with the on-demand instances. This is a 31% and a 8.3% cost saving compared to the on-demand and reserved instance based fixed provisioning approach respectively. Even though we could not test the resource configuration that supports the original throughput spike, from other experimental results, it is likely that the configuration produced by our middleware would be able to successfully satisfy the demand including the spike.

VII. RELATED WORK

Van den Bossche et al. apply linear programming to cost-optimal resource allocation in hybrid clouds [12]. In [12], they reported they experienced large variance of problem solving times. To overcome that issue, they also proposed heuristics to be able to tackle larger scale problems [13]. In both papers, they characterize workloads by CPU and memory; however, we use throughput and associated WECU units for characterizing workloads and VM instance types.

Both Vecchiola et al. and Calheiros et al. present deadline-driven resource provisioning on hybrid clouds with the Aneka middleware [14], [15]. Just as the algorithm presented in this paper, both of their algorithms keep track of the number of finished tasks and estimate the required resources to finish all the tasks before the deadline. Unlike our dynamic programming based approach, their algorithms do not take into account cost-optimality. Also, they quantify the amount of resources in a general way and do not evaluate the computational power of each instance type beforehand.

The performance stability of public cloud VM instances has been analyzed. Dejun et al. report that CPU and disk I/O performance on Amazon EC2 `m1.small` instances are relatively stable for long-term average, with up to 8% variance. However, they find that the performance of different VM instances of the same instance type is at most four times different [16]. In [17], Iosup et al. compare the performance of VM instances from four different IaaS cloud service

providers and find that Amazon's `m1.xlarge` shows the smallest performance variance whereas GoGrid's `GG.large` and ElasticHosts' `EH.small` instances have up to two times performance difference.

VIII. Conclusion and Future Work

We introduced the concept of Workload-tailored Elastic Compute Unit (WECU) as a measure of computing resources analogous to Amazon EC2's ECUs. We present a dynamic programming-based scheduling algorithm to select resources which satisfy the desired throughput in a cost-optimal way. Using a loosely-coupled benchmark running on a hybrid cloud environment, we confirmed WECUs have 24% better runtime prediction ability than ECUs on average. Moreover, simulation results with a real workload distribution of Wikipedia web service requests show that our WECU-based algorithm reduces costs by 8-31% compared to a conservative fixed provisioning approach.

Through the experiments, it is suggested the ECU-based approach needs a significantly higher over-provisioning rate to satisfy service demands compared to the one needed by the WECU-based approach. This means that the ECU-based approach has a wider search space of over-provisioning rates than the WECU-based approach does, therefore it is not easy for the ECU-based to find the right over-provisioning rate.

We have evaluated both WECU and ECU approaches using the `Trap` application only. Meanwhile, basic assumption of the WECU-based approach is that we can get higher throughput with more WECUs. Bag of tasks applications and the `Trap` application satisfies this assumption; however, for example, tightly-coupled applications may not satisfy this assumption depending on the network topology they work on. If their throughput is bounded by the network performance, adding more VM instances may not help improve the throughput. We plan to apply the WECU-based algorithm to more types of applications and further confirm applicability.

We noticed our dynamic programming based algorithm is in favor of choosing {`m1.small`, `c1.medium`, `c1.xlarge`} over {`m1.medium`, `m1.large`, `m1.xlarge`, `m3.xlarge`, `m3.2xlarge`} instance types. This happens because, as shown in Table II, most of the former group of instance types are better in Price/WECU ratio than the latter. If a workload is memory or I/O-intensive and `m1` and `m3` instance types perform better in such workloads, we expect that this will be reflected in the WECU relative performance, and price/performance ratio will make them favorable. Our performance measurement and the WECU-based resource configuration methods assume that instances created from the same instance type have the same performance. By considering individual variability of VM instance types, our proposed method could be further improved. Also, our current algorithm is cost-optimal for a given computational power, but it is not Pareto-optimal. We plan to modify our resource management algorithm to produce Pareto-optimal resource configurations.

Future work also includes implementing a budget-constrained resource management algorithm and evaluating our middleware's adaptability with real applications that have dynamically changing workloads.

Acknowledgements

This research is partially supported by Air Force Office of Scientific Research Grant No. FA9550-11-1-0332, a Yamada Corporation Fellowship, and an Amazon AWS in Education Research Grant.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep.*, vol. UCB/EECS-2009-28, Feb 2009.

[2] Amazon Web Services, "Amazon EC2 FAQ," http://aws.amazon.com/ec2/faqs/.

[3] S. Imai, T. Chestna, and C. A. Varela, "Elastic scalable cloud computing using application-level migration," in *5th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2012)*, Chicago, Illinois, USA, November 2012.

[4] C. A. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, vol. 36, no. 12, pp. 20–34, Dec. 2001.

[5] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[6] C. A. Varela, *Programming Distributed Computing Systems: A Foundational Approach*. Cambridge, MA, USA: MIT Press, May 2013.

[7] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 241–299, 2000.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[9] O. Sievert and H. Casanova, "A simple mpi process swapping architecture for iterative applications," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 341–352, 2004.

[10] T. Desell, K. E. Maghraoui, and C. A. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, pp. 323–337, June 2007.

[11] Domas Mituzas, "Wikipedia article traffic statistics," http://stats.grok.se/en/201306/cloud%20computing.

[12] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 228–235.

[13] ——, "Cost-efficient scheduling heuristics for deadline constrained workloads on hybrid clouds," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. IEEE, 2011, pp. 320–327.

[14] C. Vecchiola, R. N. Calheiros, D. Karunamoorthy, and R. Buyya, "Deadline-driven provisioning of resources for scientific applications in hybrid clouds with Aneka," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 58–65, 2012.

[15] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid clouds," *Future Generation Computer Systems*, vol. 28, no. 6, pp. 861–870, 2012.

[16] J. Dejun, G. Pierre, and C.-H. Chi, "EC2 performance analysis for resource provisioning of service-oriented applications," in *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*. Springer, 2010, pp. 197–207.

[17] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 931–945, 2011.