Energy-Efficient Resource Allocation and Provisioning Framework for Cloud Data Centers

Mehiar Dabbagh, Bechir Hamdaoui, Mohsen Guizani† and Ammar Rayes‡
Oregon State University, Corvallis, OR 97331, dabbaghm,hamdaoub@onid.orst.edu
† Qatar University, mguizani@ieee.org
‡ Cisco Systems, San Jose, CA 95134, ‡ rayes@cisco.com

***Abstract*—Energy efficiency has recently become a major issue in large data centers due to financial and environmental concerns. This paper proposes an integrated energy-aware resource provisioning framework for cloud data centers. The proposed framework: i) predicts the number of virtual machine (VM) requests, to be arriving at cloud data centers in the near future, along with the amount of CPU and memory resources associated with each of these requests, ii) provides accurate estimations of the number of physical machines (PMs) that cloud data centers need in order to serve their clients, and iii) reduces energy consumption of cloud data centers by putting to sleep unneeded PMs. Our framework is evaluated using real Google traces collected over a 29-day period from a Google cluster containing over 12,500 PMs. These evaluations show that our proposed energy-aware resource provisioning framework makes substantial energy savings.**

***Index Terms*—Energy Efficiency, Cloud Computing, Data Clustering, Workload Prediction, Wiener Filtering.**

I. Introduction

Energy efficiency has become a major concern in large data centers. According to [1], U.S. data centers consumed about 1.5% of the total generated electricity in U.S. in 2006, an amount that is equivalent to the annual energy consumption of 5.8 million households. This consumption is also expected to increase as data centers are anticipated to grow both in size and in numbers. A recent study by Cisco predicts that cloud traffic will grow 12-fold by 2015 [2]. There are also increasing environmental concerns to reduce energy consumption in industry after reporting that Information and Communication Technology (ICT) is responsible for about 2% of the global carbon emissions, equivalent to aviation [3]. All of these factors have alerted researchers to the importance of finding efficient solutions to save energy in data centers.

Cloud centers are examples of such large data centers. They often consist of a large number of servers also called physical machines (PMs). These PMs are grouped into multiple management units called clusters. Each cluster manages and controls a large number of PMs, typically in the order of thousands. A cluster can be homogeneous in that all of its managed PMs are identical, or it could be heterogeneous in that it manages PMs with different resource capacities.

Cloud providers offer these computing resources as a service for their clients and usually charge them based on their usage in a pay-as-you-go fashion. Cloud clients submit requests to the cloud provider, specifying the amount of resources they need to perform certain tasks. Upon receiving a client request, the cluster scheduler allocates the demanded resources to the client and assigns them to a PM. The virtualization technology allows the scheduler to assign multiple requests possibly coming from different clients to the same PM. Client requests are thus referred to as virtual machine (VM) requests. Cloud centers need then to support on-demand, dynamic resource provisioning, where clients can, at any time, submit VM requests specifying any amount of resources. It is this dynamic provisioning nature of computing resources that makes the cloud computing concept a great one [4]. Such a flexibility in resource provisioning gives rise, however, to several new challenges in resource management, task scheduling, and energy consumption, just to name a few [5, 6].

Energy consumption is of a special concern to cloud providers. According to a Google study [7], idle servers consume around 50% of their peak power. To save power, it is therefore important to switch servers to the sleep mode when they are not in use. This requires the development of novel techniques that can monitor PMs and effectively decide whether and when they need to be put in sleep mode.

Towards the goal of creating an energy-aware cloud, we propose an integrated resource provisioning framework that i) predicts the number of future VM requests along with the amount of CPU and memory resources associated with each of these requests, ii) provides accurate estimations of the number of PMs that the cloud center needs in the future, and iii) makes intelligent power management decisions that reduce energy consumption. Our framework is evaluated using real Google traces [8] collected over a 29-day period from a Google cluster containing over 12,500 PMs.

Our proposed energy-aware resource provisioning framework is simple, adaptive, and effective in that it does not require heavy calculations, yet provides very accurate workload predictions and makes substantial energy savings in cloud centers. In addition, it requires minimum storage capacity for storing traces needed to train the developed models. To sum up, our main contributions in this work are as follows. We:

- develop a prediction approach that combines machine learning clustering and stochastic theory to predict both the number of VM requests and the amount of cloud resources associated with each request.
- propose adaptive enhancements to our predictor that make the prediction parameters tunable in realtime based on the actual request load. This increases the prediction accuracy over time and avoids the need for frequent model training that other machine learning approaches, such as Neural Network, suffer from.

This work was supported in part by Cisco (CG-573228) and National Science Foundation (CAREER award CNS-0846044).

- propose an integrated resource provisioning framework that relies on the proposed prediction approaches to make suitable energy-aware resource management decisions.
- use real Google data traces to evaluate the effectiveness of our framework. These traces are collected from a heterogeneous Google cluster that contains more than 12K PMs.

The remainder of the paper is organized as follows. Section II reviews the related work. Section III briefly describes the different components of our proposed framework. Section IV presents the Google data used in this work, as well as the workload clustering approach. Section V presents the proposed prediction approach. Section VI presents enhancements to the proposed prediction approach. Section VII presents our power management heuristic. Section VIII evaluates the performance of our integrated resource provisioning framework. Finally Section IX concludes the paper and presents our future work.

II. Related Work

Previous work on energy savings in cloud centers targeted two levels: i) server level, where the focus is on minimizing energy consumption of each server separately; and ii) data center level, where the focus is on efficiently orchestrating the pool of servers.

A. Server Level Techniques

Many energy efficient techniques initially designed to extend battery lifetime of laptops have also been applied to save energy in general-purpose servers. These techniques target reducing energy at all computer layers: hardware, OS, compiler, and application. Due to space limitation, we only cover here OS-layer related approaches, as they involve some workload prediction relevant to our work. Readers interested in the work done on the other layers may refer to [9].

The basic idea behind OS-layer-based power management approaches is to save energy by switching idle devices, such as hard disk, to lower energy states whenever possible. The problem here is that because energy overhead due to switching the device to a lower or upper state is relatively high [10], it is worth switching a device to a lower state only when the device remains idle during a time period long enough to compensate for switching overhead. Researchers suggest first using a time-out approach [11], where a device is switched to an idle state only when it remains idle beyond a certain time-out threshold. A main disadvantage of such a simple approach is that it does not switch the device to a lower state until the time-out period has passed, resulting in not saving any energy during that period. To address this issue, researchers suggest then to use more advanced techniques based on machine learning [12, 13] to predict the idle time period length and rely on those predictions to make power management decisions.

Although many prediction approaches are proposed at the OS layer, they cannot be applied directly to predict VM requests. What these approaches predict is the length of the idle time whereas in cloud centers, multiple related parameters need to be predicted such as the number of arriving VM requests, the requested CPU and memory resources, etc. Furthermore, the proposed OS-layer related techniques study workload variations of a single device (e.g., hard disk) with the objective of saving energy in a single PM component. Unlike these techniques, we study workload variations of a cloud cluster with the goal of turning an entire PM to a lower state. It is worth mentioning that OS power management techniques are complementary to our work, as they can be applied on top of our framework to manage the different components of the active PMs.

B. Data Center Level Techniques

Energy awareness in data centers have focused on two aspects: VM consolidation and cluster-level power management.

1) VM Consolidation: Researchers investigated first where to place the received VM requests within the cloud cluster using the least number of ON PMs. This problem is similar to the standard Bin Packing (BP) optimization problem, which views VMs as objects and PMs as bins and where the objective is to pack these objects in as few bins as possible. The problem is known to be NP-hard [14], and thus approximation heuristics, such as First Fit Decreasing (FFD) and Best Fit Decreasing (BFD) have been proposed [15, 16] instead. Although these heuristics provide very close approximations to the optimal placement solution, they assume that all coming VM requests are known ahead of time, which is not the case for the on-demand computing paradigm. When requests are not known ahead of time, online BP heuristics, such as First Fit, Next Fit, and Best Fit have been proposed [17–19] to be used instead. The main problem with those online heuristics is that they don't anticipate the future workload and make VM hosting and PM switching decisions based only on the currently received VM requests. Unlike those heuristics, our framework takes a further step by predicting the VM requests to be received in future and relies on that to make intelligent power management decisions.

Given that VM requests can be terminated any time the cloud client wants, it is then highly likely that PMs become under-utilized over time, resulting in inefficient use of data center resources. Migration and dynamic consolidation techniques have been proposed as key solution approaches for improving datacenter resource efficiency [20–25]. For example, dynamic consolidation approaches proposed in [20–23] allow VMs to migrate from the under-utilized PMs so that the workload can be consolidated on a smaller subset of PMs, allowing further PMs to be turned to sleep. The authors in [20, 21] considered the performance degradation associated with the migration process and proposed a dynamic consolidation framework that guarantees certain SLA level. The work in [24] took the migration energy overhead into account when making such decisions. All the mentioned dynamic consolidation techniques are complimentary to our work as they can be applied on top of our framework to pack the already scheduled VMs more tightly over time. Our work is different as we are predicting the number and the amount of VM requests to be received in the future. In fact, these predictions can improve the migration decisions made by those dynamic consolidation techniques by considering not only the current hosted requests, but also the future requests predicted by our framework when making their migration decisions.

2) Cluster-Level Power Management: The work in [26, 27] exploited the fact that servers in cloud centers are distributed in different geographical locations with varying electricity prices and proposed different strategies for placing the received requests on these distributed servers such that the energy costs are mini-

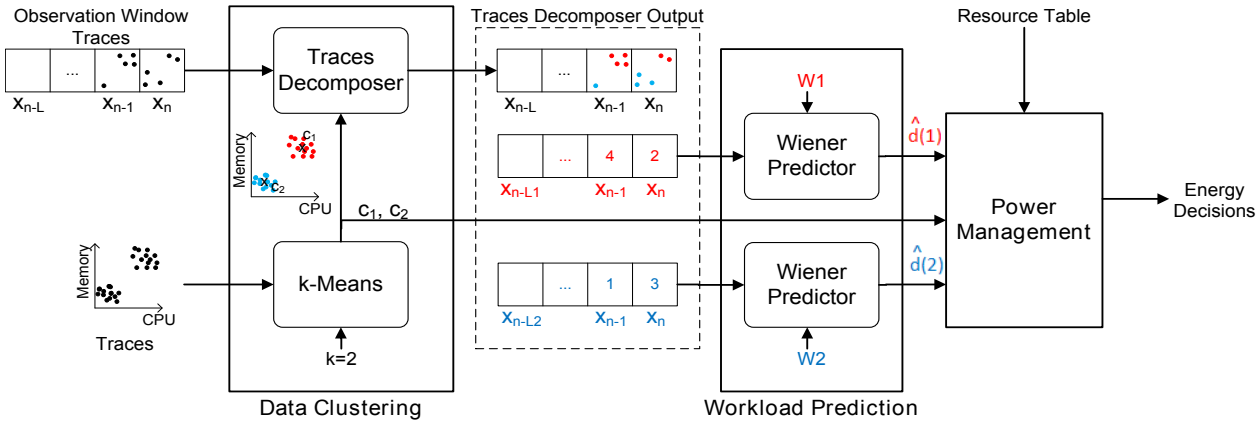

Fig. 1. Flow chart of the proposed framework

mized. Although these techniques reduce the cluster's electricity bills, they do not actually reduce the consumed energy.

Power management approaches that save energy by adjusting CPU's operating frequency and voltage of PMs based on their workload within the cluster have also been proposed [28, 29] (these are known as Dynamic Voltage and Frequency Scaling (DVFS)). Unlike these techniques, in our framework redundant PMs are completely turned to sleep mode instead of reducing their operating frequency and voltage, thereby achieving higher energy savings.

Several predictive frameworks were proposed in [30–35] to reduce the number of ON PMs by tuning the allocated resources of the already scheduled VMs. The work in [30–32] considers the case where a client requests multiple VMs to run a certain application. Rather than reserving a fixed number of VMs for each application all the time, the authors dynamically adjust this number based on predicting the application's demands. PRESS [33] on the other hand performs *VM resizing* where the allocated resources (e.g. CPU or memory) for each scheduled VM are tuned based on predicting the client's demands. The authors in [34, 35] add dynamic consolidation on top of VM resizing and thus use VM migration to compact the resized VMs on fewer ON PMs. Our framework is different from those predictive frameworks as we are predicting the number of VM requests to be received from all the cloud clients in addition to the resource demands associated with each request. These predictions are used later to estimate the number of ON PMs needed to support the future workload. This is different from predicting the number of needed VMs for a specific application that was already scheduled on the cluster [30–32], or the future resource demands of an already scheduled VM [33–35].

There have been many approaches proposed to predict the coming load in distributed systems, such as data centers and grid systems. Hidden Markov models [36] and polynomial fitting [37] are examples of workload prediction methods that have been used in Grid systems. The fact that cloud workloads are made up of requests with multiple resource demands makes such predictions schemes not applicable to the cloud paradigm.

III. The Proposed Framework

Our framework has three major components: data clustering, workload prediction, and power management. In this section, we briefly describe these components so as the reader will have a global picture of the entire framework before delving into the details. Detailed descriptions are provided in later sections. Throughout this section, we refer to Fig. 1 for illustration.

A. Data Clustering

Our developed prediction approach relies on observing and monitoring past workload variations during a time period referred to as the *observation window* in order to predict the workload coming in a certain future period referred to as the *prediction window*. A VM request requested by a cloud client typically consists of multiple cloud resources (e.g., CPU, memory, bandwidth, etc.). This multi-resource nature of the requests poses unique challenges when it comes to developing prediction techniques. Also, different cloud clients may request different amounts of the same resource. Therefore, it is both impractical and too difficult to predict the demand of each type of resource separately, as ideally this is what is needed to be able to make optimal power management decisions. To address this issue, we instead divide VM requests into several categories, and then develop prediction techniques for each of these categories. This is known as clustering.

1) k-Means clustering: Our first step is then to create a set of clusters to group all types of VM requests; i.e., each VM request is mapped into one and only one cluster, and all requests belonging to the same cluster possess similar characteristics in terms of their requested resources.

In the general case, each VM request is associated with d types of resources such as CPU, memory, bandwidth, etc. In order to divide these requests into multiple categories, we represent first each request as a point in the $\mathbb{R}^d$ space. As for the Google cluster [8], only two types of resources, CPU and memory, are associated with each request. Thus these requests are represented in the $\mathbb{R}^2$ space, where each point is a request and the two dimensional coordinates of the point are the amount of CPU and memory resources associated with the request. Clustering these data points into a number of clusters is done using k-Means algorithm [38].

As shown in Fig. 1, the k-Means algorithm takes as an input the Google traces and the number of clusters, k, and outputs k clusters, each specified by its center point. For the case of the Google data, where requests have two types of resources (CPU

and memory), the cluster centers are points in the $\mathbb{R}^2$ space. In the general case, when requests have d types of resources, the cluster centers would be points in the $\mathbb{R}^d$ space. For illustration purposes only, Fig. 1 shows an example with $k = 2$, where two clusters produced by the algorithm are represented in Red and Blue, as shown in the Cluster Groups graph. Note that the parameter k needs to be chosen *a priori* and given as an input to the clustering algorithm. In Section IV-B, we show how such a parameter is selected.

It is worth mentioning that the clustering stage is done only once on large traces collected from the cloud center. The clustering stage can be launched again if the request characteristics change significantly in the cloud center over time. Our experiments on the Google traces show that running the clustering phase only once on a large training data is enough to extract the characteristics of the submitted VM requests.

2) Traces Decomposer: Once the k clusters and their center points are determined, they are given as an input to the Traces Decomposer module (shown in Fig. 1), which is responsible for mapping each request received during the observation window into one cluster. The observation window is split into $L+1$ time slots, $n, n-1, \ldots, n-L$, as follows. Suppose a prediction needs to be made at time n. In this case, slot n corresponds to time interval $[n-10, n]$ (in seconds); slot $n-1$ corresponds to time interval $[n-20, n-10]$, slot $n-i$ corresponds to time interval $[n-10(i+1), n-10i]$, and so on. The Traces Decomposer tracks the number x_{n-i} of received requests in time slot $[n-10(i+1), n-10i]$ of the observation window for all $i = 0, 1, \ldots, L$, and maps each request within the slots into one cluster.

B. Workload Prediction

We use stochastic Wiener filter prediction to estimate the workload of each category/cluster. The Stochastic Predictor, as shown in Fig. 1, is made of k Wiener filters. Each filter takes as an input the number of received requests for a certain category during the observation window, and uses it to predict the number of requests of that same category to arrive in the prediction window. This makes the problem easier to solve as there are infinite number of possibilities for the amount of CPU and memory resources that a client may request.

The reasons for choosing Wiener filter as a predictor are: First, it outperforms the other schemes in terms of prediction accuracy as will be seen in Section VI-B. Second, it is simple, as the prediction for each category is a weighted summation of the recently observed number of requests of that category. Third, it has a sound theoretical basis. Finally, it is easy to improve the original Wiener filter to support online learning, making it more adaptive to changes in workload characteristics. This is done by updating the filter's weights as new observations are made over time without requiring heavy calculations or large storage space as will be seen in Section VI-A.

The parameters that need to be determined for each filter branch are: the length of the observation window, the length of the prediction window, and the weights. These parameters are determined in Section V.

C. Power Management

The predictions of all categories along with their center points are all next passed to the Power Management module, which uses this information to decide which PMs need to go to sleep and which ones need to be kept ON. This unit keeps track of all PMs in the cloud cluster and stores their current utilizations and states (ON or sleep). It uses a modified Best Fit Decreasing (BFD) heuristic to fit the predicted VM requests in PMs in order to determine how many ON PMs will be needed in the coming prediction window.

The original BFD algorithm [16] tries to pack VM requests in the fullest PM with enough space. In order to do that, it sorts PMs from the fullest to the least full and iterates over the ordered list of PMs trying to pack the VM request within the first PM that has enough space. The original algorithm could not be applied directly in our framework as it considers only one dimension and thus a modification is needed to make the algorithm work for the case where PMs and VMs have multiple dimensions (i.e. multiple types of resources). This limitation has been addressed by mapping these multiple dimensions into a single metric that combines them all. Examples of such a metric include taking the sum or the product of those multiple dimensions. In our heuristic, we considered the product metric, as our experiments show that it outperforms the summation metric[1]. Furthermore, our proposed heuristic takes the energy efficiency problem into account when sorting the PMs as it sorts PMs by the following criteria (in ascending order):

(i) PMs that are ON
(ii) PMs that have higher utilizations. The utilization metric is defined as the product of the CPU and the memory capacities of the PM.
(iii) PMs that have higher capacities. Similarly, the capacity metric is defined as the product of the memory and CPU capacities of the PM.

The intuition behind our sorting criteria is as follows: we want to make use of the available ON PMs, which already have some scheduled VMs; so ON PMs are ranked first. We then use the utilization metric as the next sorting criterion, since increasing the utilization of the PMs makes the cluster as a whole more energy-efficient, as it results in switching to sleep more PMs. Finally, PMs are sorted based on their capacities, as one can fit more VMs within a PM when PMs are of large capacities. Detailed description of the proposed energy saving heuristic is provided in Section VII.

IV. Data Clustering

In this section, we first begin by presenting the Google data traces that we used in this work to train and test our developed energy-aware resource provisioning models, and then present our workload clustering and classification findings.

A. Google Traces

We conduct our experiments on real Google data [8] that was released in November 2011 and consists of a 29-day traces collected from a cluster that contains more than 12,500 PMs. A summary of this data is provided in Table I.

The cluster is heterogeneous as it supports different PM configurations, as described in Table II. The first column shows the number of PMs in the cluster whose configurations are

[1]Other combining metrics will be investigated in the future.

specified in the subsequent columns. The second column shows the architecture type of these PMs. Note that there are three different types of architecture, referred to as A, B and C, as their actual type has been obfuscated for privacy reasons. PMs from the same architecture may have different CPU and memory capacities as shown in column three and four, respectively. These capacities are normalized to the maximum capacity also for privacy reasons, and thus the reported capacities are all less than or equal to one.

The traces provided by Google are collected at the cluster level, where VM requests are submitted and scheduled, and at the PM level too, where the amount of utilized resources are tracked over time for each VM. Previous work on this data has focused on studying general statistical characteristics of the cloud cluster [39, 40] or on classifying VMs into a number of groups based on the amount of resources they utilize over time [41].

TABLE I
CHARACTERISTICS OF GOOGLE TRACES

Trace Characteristic	Value
Duration of Traces	29 days
Number of PMs	12,583
Number of VM requests	> 50M
Compressed size of data	39GB

TABLE II
CONFIGURATIONS OF THE PMS WITHIN THE GOOGLE CLUSTER

Number of PMs	PM Configurations		
	Architecture	CPU	Memory
6732	A	0.50	0.50
3863	A	0.50	0.25
1001	A	0.50	0.75
795	C	1.00	1.00
126	B	0.25	0.25
52	A	0.50	0.12
5	A	0.50	0.03
5	A	0.50	0.97
3	C	1.00	0.50
1	A	0.50	0.06

The clustering step implemented in our framework is different from the work in [41], as VM requests are clustered into multiple categories based on the amount of requested CPU and memory, rather than classifying these requests based on the amount of resources they utilize over time. To the best of our knowledge, this is the first work that considers developing a workload prediction approach based on this data.

Our experiments utilize the data provided in the task event table, where each VM request is called a task and each VM submission/termination request is referred to as an event. It is worth mentioning that Google chooses to allocate containers rather than full virtual machines for these tasks. Each one of these two choices has their advantages and disadvantages. Full machine virtualization offers greater isolation at the cost of greater overhead, as each virtual machine runs its own full kernel and operating system instance. Containers, on the other hand, generally offer less isolation but lower overhead through sharing certain portions of the host kernel and operating system instance. As far as our framework is concerned, we are predicting the future workload by predicting the number of task requests and amount of resources associated with these requests. These tasks could be handled by either full virtualization or by containers but this does not affect the applicability of our framework in predicting the future workload and in estimating the amount of needed PMs for the future workload. Throughout the paper, we refer to these tasks as VMs. A detailed description of the data is provided in [42], and the following are the features used by our framework:

- VM ID: a unique identifier for each VM. Kept anonymous and replaced by its hash value.
- Client ID: a unique identifier for each cloud client. Kept anonymous and replaced by its hash value.
- Event type: specifies whether the event is a submission or a release request. It is worth noting that clients may submit or release a VM request whenever they desire.
- Timestamp: time at which the event happened.
- Requested CPU: amount of requested CPU.
- Requested memory: amount of requested memory.

Note that the requested amount of CPU (memory) resources is always between 0 and 1 as it is normalized to the PM with the largest CPU (memory) capacity in the Google cluster. Also note that clients do not necessarily use all requested resources at all the time, as their usage varies depending on their needs. Yet these resources are reserved and allocated for them for so long as their requests are not released, regardless of whether they are using them or not.

Since the size of the Google data is huge, we use two chunks of the traces to tune the different parameters that are involved in our framework. We refer to these chunks as the training set and the validation set. The training set includes the Google traces collected during a 24-hour period and the validation data set contains the traces collected during a 16-hour period. A separate chunk of the Google data called the testing set with a duration of 29 hours is used later to estimate the accuracy, the energy savings and the utilization that our framework achieves. The testing chunk is only used to evaluate our framework's performance on unseen data and no parameters are selected based on this chunk.

B. Workload Clustering

We use k-Means [38], a well-known unsupervised learning algorithm, to cluster VM requests into k categories. The k-Means algorithm assigns N data points to k different clusters, where k is *a priori* specified parameter. The algorithm starts by an initialization step where the centers of the k clusters are chosen randomly, and then assigns each data point to the cluster with the nearest (according to some distance measure) center. Next, these cluster centers are recalculated based on the current assignment. The algorithm repeats i) assigning points to the closest, newly calculated clusters and then ii) recalculating the new centers until the algorithm converges (no further changes occur).

As mentioned previously, each VM request is mapped into $\mathbb{R}^2$ where the two dimensions are the requested amount of CPU and memory. The Euclidean distance is used as the measure of distance between the points and the centers. k-Means is run for 20 different initial centers and the resulting clusters with the lowest error are reported.

One of the important parameters that needs to be tuned when using k-Means is k, the number of clusters. A heuristic approach is implemented to tune this parameter, in which the Sum of Squared Distances (SSD) is plotted as a function of the number

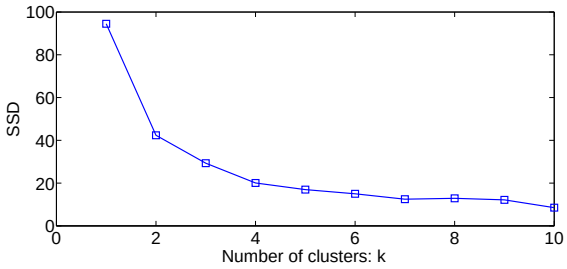

Fig. 2. The elbow criteria for selecting k.

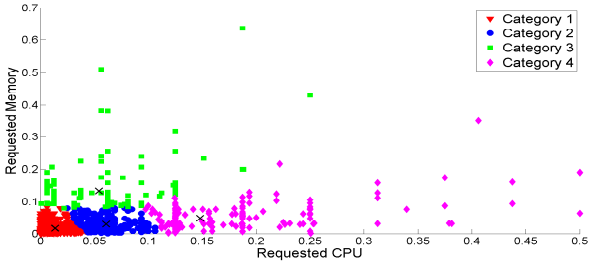

Fig. 3. The resulting four clusters/categories for Google traces.

of clusters k. SSD represents the clustering error when each point in the data set is represented by its corresponding cluster center, and is mathematically equal to $\sum_{i=1}^{k}\sum_{r\in\mathcal{C}_i}\mathrm{d}(r,c_i)^2$ where $\mathcal{C}_i$ denotes cluster i; i.e., set of all points belonging to the i^{th} cluster, c_i denotes cluster i's center point, and $\mathrm{d}(r,c_i)$ is the Euclidean distance between r and c_i.

Fig. 2 shows SSD as a function of the number of clusters k plotted based on the training set of the Google traces. Note that as k increases, SSD decreases monotonically, and hence so does the clustering error. Recall that while increasing the value of k reduces the clustering error, it also increases the overhead incurred by the prediction technique (to be presented in next sections), since a predictor branch needs to be built for each cluster/category. For this, the heuristic approach searches then for the "elbow" or "knee" point of the plot, which is basically the point that balances between these two conflicting objectives: reducing clustering errors and maintaining low overhead. As can be seen from Fig. 2 which is based on the training traces, the value 4 for k strikes a good balance between accuracy and overhead. Hence, in what follows we use $k = 4$.

Fig. 3 shows the resulting clusters for $k = 4$ based on the training set, where each category is marked by a different color/shape and the centers of these clusters c_1, c_2, c_3 and c_4 are marked by 'x'. Category 1 represents VM requests with small amount of CPU and small amount of memory; Category 2 represents VM requests with medium amount of CPU and small amount of memory; Category 3 represents VM requests with large amount of memory (and any amount of requested CPU). Category 4 represents VM requests with large amount of CPU (and any amount of requested memory). Observe from the obtained clusters that requests with smaller amount of CPU and memory are denser than those with large amounts.

V. Workload Prediction

In this section, we determine and estimate the parameters of the proposed prediction approach.

A. Length of the Prediction Window

An important parameter that needs to be estimated for the stochastic predictor is the length of the prediction window, T_p. This represents the length of the time period in the future for which the workload needs to be predicted in order to decide whether or not PMs need to be switched to sleep mode. Letting P_{idle} denote the power the PM consumes while in ON and idle, the amount of energy that the PM consumes if it is left ON and idle during T_p is $E_{idle} = P_{idle} \times T_p$. If we decide to switch the PM to the sleep mode, then the consumed energy $E_{sleep} = E_o + P_{sleep} \times (T_p - T_o)$, where P_{sleep} is the consumed power when in the sleep mode; E_o is the transition energy, equaling the energy needed to switch the machine to the sleep mode plus the energy needed to wake up the machine later; and T_o is the transitional switching time.

Let T_{be} be the amount of time during which keeping the PM ON and idle consumes an amount of energy that is equal to the energy consumed due to mode transition plus that consumed while the PM is in the sleep mode during that same period. Here, T_{be} represents the break-even time, and must satisfy:

$$P_{idle} \times T_{be} = E_o + P_{sleep} \times (T_{be} - T_o)$$

Note that switching a PM to sleep mode saves energy only when the PM stays idle for a time period longer than T_{be}; that is, $T_p \geq T_{be}$ must hold in order for the power switching decisions to be energy efficient.

In this work, we rely on the energy measurement study of physical machines conducted in [43] to estimate the break-even time, T_{be}. Table III shows those measurements (reported in [43]) that are used to calculate T_{be}. These measurements are also used later to estimate the energy savings our proposed techniques achieve. Based on these measurements, our calculation yields $T_{be} = 47$ seconds; that is, T_p should be larger than 47 seconds. It is worth noting that these energy measurements were based on a certain type of commercial PMs, and hence, these numbers might slightly change depending on the type of PMs in the cluster. As mentioned before, Google does not provide information about the types of these PMs. Although we took in our experiments a conservative approach and chose $T_p = 60$ seconds in order to account for the cases where these numbers might be different for other types of PMs, our proposed approach works for any other T_p choice.

It is worth mentioning that since $T_p = 60$, the predictors in our framework are run every minute to predict the number of requests in the coming minute. The power management module relies on these predictions to estimate the number of needed PMs in the next minute. Since Switching PMs from sleep to ON takes time, then the predictors are run before the minute ends by an amount of time that is equal to the switching time so as to have the PMs ready to cover the workload in the coming minute.

TABLE III
ENERGY MEASUREMENTS NEEDED TO CALCULATE T_{be}

Parameter	Value	Unit
P_{sleep}	107	Watt
P_{idle}	300.81	Watt
P_{peak}	600	Watt
$E_{on \rightarrow sleep}$	5510	Joule
$E_{sleep \rightarrow on}$	4260	Joule
T_o	6	Seconds

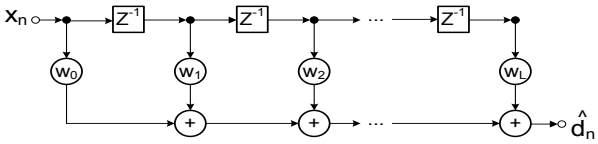

Fig. 4. The general structure of Wiener filter.

B. Weights of the Stochastic Predictor

Let n be the time at which the prediction needs to be made. The general structure of the Wiener predictor for each of the four categories can be represented as shown in Fig. 4, where:

- x_{n-i}: is the number of requests of the considered category received in the period between $n-10(i+1)$ and $n-10i$ seconds.
- d_n: the desired output of the category predictor. This represents the actual number of requests for the considered category in the coming prediction window.
- $\hat{d}_n$: is the predicted number of requests for the considered category in the coming prediction window.
- L: is the number of taps that the predictor relies on in making predictions.
- w_i: is the i$^{\text{th}}$ tap's weight.

Wiener filter predicts the future requests assuming x_n is a wide-sense stationary process. The predicted number of requests, $\hat{d}_n$, is a weighted average of the previous observed requests and thus can be written as $\hat{d}_n = \sum_{i=0}^{L} w_i x_{n-i}$. The prediction error, e_n, can be calculated as the difference between the actual and predicted number of requests; i.e.,

$$e_n = d_n - \hat{d}_n = d_n - \sum_{i=0}^{L} w_i \, x_{n-i}$$

The objective is to find the weights that minimize the Mean Squared Error (MSE) of the training data, which is:

$$MSE = E[\, e_n^2 \,] \tag{1}$$

MSE represents the second moment of error, and the reason we chose it as an objective function is due to the fact that it minimizes both the average error and the variance of error putting more weight on the average error ($E[e_n^2] = (\, E[\, e_n \,])^2 + var(\, e_n \,)$).

Differentiating Eq. (1) with respect to w_k and setting this derivative to zero yields, after some algebraic simplifications,

$$E\big[d_n \, x_{n-k}\big] - \sum_{i=0}^{L} w_i E\big[x_{n-k} \, x_{n-i}\big] = 0$$

By letting

$$R_{dx}(k) = E\big[d_n \, x_{n-k}\big] \tag{2}$$
$$R_{xx}(i-k) = E\big[x_{n-k} \, x_{n-i}\big] \tag{3}$$

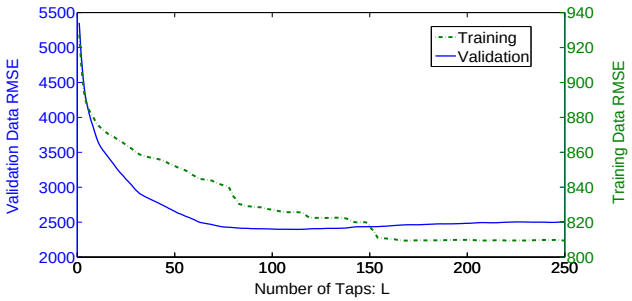

Fig. 5. RMSE for 3rd category predictor

it follows that $R_{dx}(k) = \sum_{i=0}^{L} w_i R_{xx}(i-k)$.

Similar equations expressing the other weights can also be obtained in the same way. These equations can be presented in a matrix format as $R_{dx} = R_{xx} W$, where

$$R_{xx} = \begin{bmatrix} R_{xx}(0) & R_{xx}(1) & \dots & R_{xx}(L) \\ R_{xx}(1) & R_{xx}(0) & \dots & R_{xx}(L-1) \\ \vdots & \vdots & \ddots & \vdots \\ R_{xx}(L) & R_{xx}(L-1) & \dots & R_{xx}(0) \end{bmatrix}$$

$$W = \begin{bmatrix} w_0 & w_1 & \dots & w_L \end{bmatrix}^T$$

$$R_{dx} = \begin{bmatrix} R_{dx}(0) & R_{dx}(1) & \dots & R_{dx}(L) \end{bmatrix}^T$$

Given R_{xx} and R_{dx}, the weights can then be calculated as:

$$W = R_{xx}^{-1} R_{dx} \tag{4}$$

We rely on the training data set to calculate R_{xx} and R_{dx} for each category. To estimate these parameters for a certain category, we divide the training data into N slots where the duration of each slot is 10 seconds. We first calculate the number of requests of the considered category that are received in each slot. Then, we calculate the elements of R_{xx} using the unbiased correlation estimation as:

$$R_{xx}(m) = \frac{1}{N-m} \sum_{j=0}^{N-m-1} x_{j+m} \, x_j \tag{5}$$

The elements of R_{dx} can also be estimated using the correlation coefficients. Since d_n represents the number of requests in the coming prediction window which has a duration of 60 seconds, we can write $d_n = \sum_{i=1}^{6} x_{n+i}$. Plugging the expression of d_n in Eq. (2) yields the correlations that can be estimated from the training data. An estimation of the weight vector follows then for each category predictor provided R_{dx} and R_{xx} are known. These weights lead, in turn, to the lowest MSE for the training data.

C. Length of the Observation Window

The last parameter that needs to be tuned for each category predictor is the length of the observation window. As mentioned before, the observation window is divided into L slots or (also called) taps, each tap/slot is of length 10 seconds. This Wiener filter is referred to as an L-tapped filter. To determine L for a category predictor, we first need to find the optimal weight vectors of the Wiener predictor under different values of L on

the training data, and then test the performance of these weight vectors on the unseen validation data.

Fig. 5 shows the Root Mean Square Error ($RMSE$) of the 3rd category predictor for both the training and validation data sets for different values of L. Observe that the training data $RMSE$ decreases monotonically as L increases. To understand this, consider two Wiener filters: one with L taps and the other with $L + R$ taps. Recall that we need to find optimal weights that when multiplied by these taps leads to the minimum MSE of the training data. As a result, the model with $L + R$ taps can achieve the same accuracy on the training data set as the model with L taps by setting the weights of all the additional R taps to zero. Thus, the model with a given number of taps will, in the worst-case scenario, achieve the same accuracy as any model with lower numbers of taps. In general, models with larger numbers of taps can still find some correlations specific to the training data that lead to a better accuracy. Consequently, the training error will continue to decrease as we increase the number of taps.

However, by observing the behavior of the validation data, note that $RMSE$ decreases first until it reaches a point beyond which the error can no longer be reduced even if L is increased further. Also, observe that if we continue to increase L, the validation $RMSE$ starts to increase. This behavior is expected and is known as the overfitting phenomena. After increasing the number of taps beyond a certain limit, the model tries to find correlations between the different requests over time. These correlations are specific to the training data so we say that increasing the number of taps beyond a certain limit increases the complexity of the model and starts finding correlations that do not exist in the general traces but are specific to the training data. Based on our experiments, we chose $L = 80$, as it achieves the best accuracy on the unseen validation data, meaning that the observation window of the 3rd category predictor relies on the traces in the previous 80 taps collected in the previous $80 \times 10 = 800$ seconds.

Using a similar approach, the optimal numbers of taps for the other three categories are determined to be 20 (category 1), 80 (category 2), and 34 (category 4). Graphs for these three categories are omitted due to space limitation.

VI. Prediction Enhancements

We now describe two enhancements that can be done on the proposed stochastic predictor. These enhancements aim at improving prediction accuracy and making more efficient cloud resource allocation and management.

A. Adaptive Predictor

In our framework, we rely on the traces from the training data to calculate the weights. One of the problems that might be encountered in a practical implementation of the framework is that we will need to retrain the model to adjust these weights since the workload characteristics may vary over time. We refer to such a model that needs training from time to time as the static Wiener model. We provide in this subsection an adaptive Wiener model that increases the accuracy over time and avoids the need to store large traces to retrain the model on new training data in order to update weights.

Initially, the adaptive model predicts the number of requests in the coming minute based on the learned weights from the training stage. Next, the model observes the number of requests that were received in that minute for each category. It uses these observations to update the weights. The adaptive model continues doing this every minute. As a result, our adaptive approach uses all the traces that were observed until the time of the prediction to find the optimal weights. This is different from the static approach that relies only on a chunk of the traces from the training data to adjust these weights. The adaptive predictor has an overhead since after observing the actual workload in each minute it needs to do some calculations to update these weights. However, these updates increase the accuracy over time and makes the model dynamic to the latest variations of the workload.

As showed previously in Section V-B, we only need to calculate the correlations for different lags of time in order to calculate the weights. In order to reduce the calculation overhead in our adaptive model, we introduce two variables for each correlation coefficient that aggregate all the previous calculations. We only need to store these variables instead of storing all the previous traces and thus the amount of storage needed to update these weights is reduced. Furthermore, these variables can be updated easily once we observe the new workload which reduces the calculation overhead further.

The unbiased estimation of the coefficients given in Eq. (5) can be written as $R_{xx}(m) = Sum(m)/Counter(m)$, where $Sum(m) = \sum_{j=0}^{N-m-1} x_{j+m}\, x_j$ and $Counter(m) = N - m$ are two aggregate variables.

Recall that x is a random variable that represents the number of requests received within ten seconds. In each minute, six new observations of x take place. We refer to these new observations by the group O. The two aggregate variables, $Sum(m)$ and $Counter(m)$, are then updated as:

$$Counter(m) \leftarrow Counter(m) + 6$$
$$Sum(m) \leftarrow Sum(m) + \sum_{k \in O} x_{k-m}\, x_k$$

The new correlations can be estimated easily by dividing Sum by $Counter$ for the different lags. Next, we calculate the updated weights using Eq. (4).

Fig. 6 shows the actual number of requests received for the third category over time. We also plot the predicted number of requests based on both the static and adaptive Wiener predictors. Note that initially both adaptive and static models give the same predictions as the adaptive model still has not made many updates. Observe how the adaptive model improves over time as its predictions become closer to the actual number of requests. We evaluate the accuracy of both of these models in Section VI-B, and present the second enhancement in Section VI-C.

B. Prediction Accuracy

We now compare the accuracy of the two proposed adaptive and static Wiener predictors against one another and against six other prediction techniques, which are described next:

- **Last minute predictor:** returns the same number of requests observed during the previous minute.

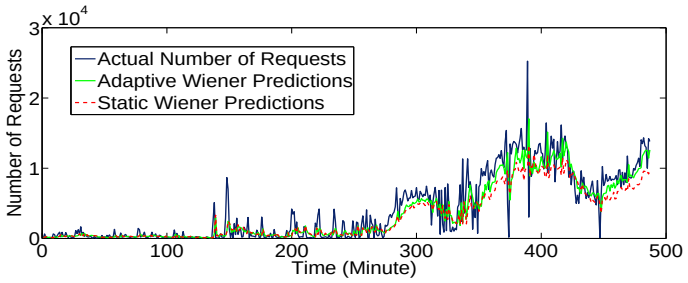

Fig. 6. Static versus adaptive Wiener filter prediction for category 3.

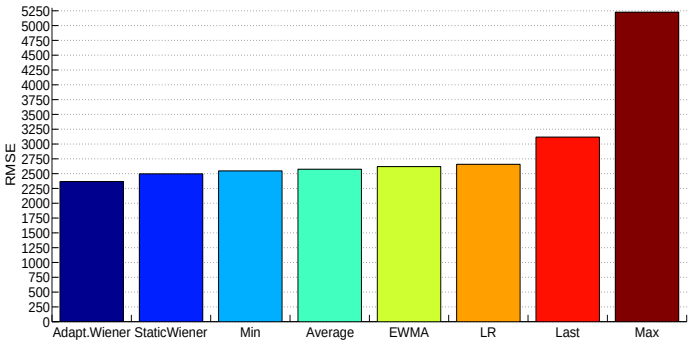

Fig. 7. RMSE comparison of different predictive approaches.

- **Min predictor:** observes the number of requests received in each minute during the last five minutes, and returns the minimum of these five observations.
- **Max predictor:** similar to Min predictor but returns the maximum instead of the minimum.
- **Average predictor:** similar to Min predictor but returns the average instead of the minimum.
- **Exponential Weighted Moving Average (EWMA) predictor:** assigns exponentially decreasing weights for the previous observations rather than equal weights as in the Average Predictor. The smoothing factor for the EWMA predictor is tuned by evaluating the predictor on the training data using different smoothing factors. The value of the smoothing factor that achieves the highest accuracy on the training data is selected for future predictions.
- **Linear Regression (LR) predictor:** observes the number of requests received in each minute during the last five minutes, and returns a weighted average of the previous observations where the considered weights are tuned based on the training data.

Fig. 7 shows the Root Mean Square Error (RMSE) for the different predictive approaches. Since in each minute we need to predict the number of requests for four categories, the reported RMSE in Fig. 7 is the summation of the RMSE for the predictions of the four categories. These evaluations are reported on a testing data of Google Traces that includes 2.5 milion VM requests received during a 29-hour period. The testing data set is different from the training and validation data sets that are used for tuning the parameters. This provides a fair comparison as it shows the performance of our predictor over new data that it did not see before.

Note also from Fig. 7 that both the static and adaptive Wiener predictors have lower RMSE than all the other predictive approaches. The adaptive Wiener predictor achieves lower RMSE than the static predictor as it updates the weights regularly based on the observations that are seen in each minute. This increases the accuracy of the predictor and makes it more adaptive to new workload changes, something that the other studied prediction techniques lack. The static Wiener predictor would have achieved an accuracy that is close to that achieved by the adaptive predictor if it was trained regularly every certain duration of time such as every two hours. It is also worth mentioning that the basic predictive techniques have also been tested for different time duration (not only 5 minutes but 10 minutes, 15 minutes,...etc.). For all of these cases, the basic approaches report a performance worse than the static and adaptive Wiener approaches. We report only the basic predictive approaches that rely on the number of requests received during the last 5 minute duration in their prediction due to space limitation.

C. Safety Margin

Our stochastic predictors may still make errors as the predictions may be larger or smaller than the actual number of requests. A main problem when we underestimate the number of future requests is that we will need some time to wake one of the sleeping machines when an unpredicted request arrives. As a result, clients may observe an undesired short delay before their resources are allocated. In order to reduce the occurrences of such cases, we add a safety margin to accommodate for such variations. The cost of this safety margin is that we will keep some redundant PMs in the working state which will reduce slightly the total energy savings.

In order to avoid the problem of selecting an appropriate static value for the safety margin, we make it dynamic and related to the accuracy of our predictors. Our proposed safety margin increases if our predictions deviate much from the actual number of requests and it decreases when accurate predictions are made. Since these deviations may vary over time, we calculate an exponentially weighted moving average (EWMA) of the deviations while giving higher weights to the most recent ones. Initially, Dev is set to zero; i.e., $Dev_0 = 0$, and for later time n, Dev_n is updated as

$$Dev_n = (1-\alpha)Dev_{n-1} + \alpha \left| d_{n-1} - \hat{d}_{n-1} \right|$$

where $0 < \alpha < 1$ is the EWMA smoothing factor used to tune between the weight given to most recent deviations over that given to the previous ones. Dev is updated before the next prediction is made by observing the deviation between the predicted and actual number of requests in the previous minute. One advantage of EWMA is that the moving average is calculated without needing to store all observed deviations.

We add the weighted average Dev to our predictions in the next minute after we multiply it by a certain parameter β. The predicted number of requests with safety margin $\bar{d}_n$ can then be calculated as $\bar{d}_n = \hat{d}_n + \beta Dev_n$.

Figs. 8 and 9 show the actual number of requests in addition to the predictions with and without safety margin for the second and third categories on a part of the testing set. Note that the predictions for the remaining two categories were not included due to space limitation. As for the safety margin parameters,

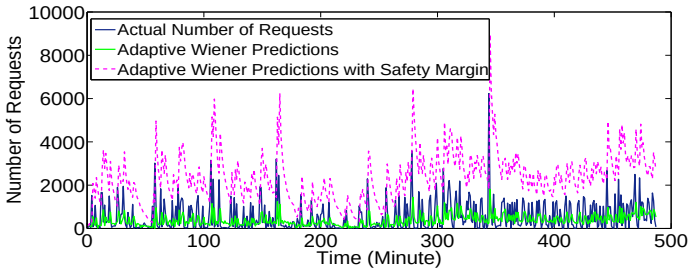

Fig. 8. Adaptive prediction with and without safety margin for category 2.

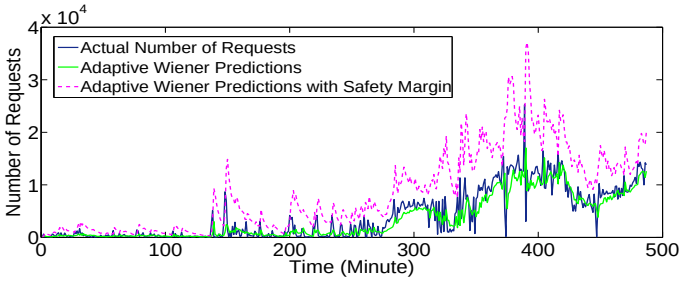

Fig. 9. Adaptive prediction with and without safety margin for category 3.

α and β are set to 0.25 and 4, respectively; these values are selected experimentally by picking the values that provide the best predictions on the training set. Note from these figures that the prediction with safety margin forms an envelope above the actual number of requests. The envelope becomes tighter when our predictions are accurate and becomes wider when our predictions deviate much from the actual number of requests for each category. When bursts in the actual number of requests occur, the envelope becomes wider directly after that burst to avoid any future possible bursts. If no other bursts are observed in the following minutes, then the envelope becomes tighter over time as the recent deviations which have higher weight are small.

We also show in Fig. 10 the total number of unpredicted PMs that were waken from sleep to accommodate the workload when our framework is evaluated on the entire testing data with and without safety margin. These numbers reflect how often clients had to wait for sleeping PMs to be switched ON before their requests got allocated. Since the Min Predictor had the least error among the approaches we compared against in Fig.7, we also report in Fig. 10 the total number of unpredicted PMs when the Min Predictor is used in our framework to make predictions. Observe that the adaptive Wiener filter without safety margin had a lower number of unpredicted PMs compared to the Min Predictor. The results also show that by adding a safety margin to the predictions of the Wiener filter, the number of unpredicted PMs that were switched ON was reduced further by 87% compared to the case without safety margin.

VII. Power Management

As mentioned previously in Section III, the Power Management module keeps track of all the PMs in the cloud cluster and stores their capacities, their current utilization and their state (ON or sleep) in a table, referred to as the `ResourceTable`. Let $\bar{d}(i)$ be the predicted number of requests of the ith category in the coming prediction window (taken from the output of the

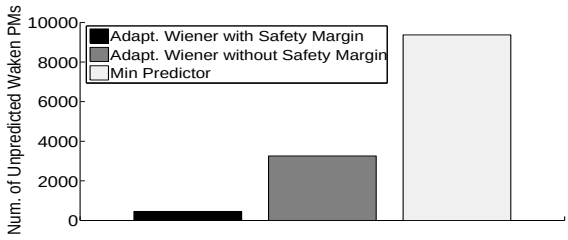

Fig. 10. Total number of unpredicted waken PMs during the entire testing period.

workload predictor); and c_i be the center of the ith category (taken from the data clustering stage). The Power Management module relies on `ResourceTable` along with the centers and the predicted number of requests for each category to decide on the number of needed PMs in the coming prediction window. Algorithm 1 shows the heuristic approach used by the Power Management module to determine the number of needed PMs. The steps of the algorithm are as follows:

In line 1, a copy of `ResourceTable` called `PMTable` is made to be used as a draft table to determine the number of needed PMs. This draft table is made so that the state and the utilization fields of the original `ResourceTable` are not changed while trying to fit the predicted VM requests in the PMs in order to estimate the number of needed PMs. In line 2, `PMTable` is sorted based on the three criteria introduced in Section III-C. In line 3, `VMList` is a list constructed to contain all the predicted requests in the coming prediction window. In fact, `VMList` contains $\bar{d}(1)$ VM request with the CPU and memory resources specified by c_1, $\bar{d}(2)$ VM request with the resources specified by c_2 and so on for all the categories. In line 4, the requests in `VMList` are sorted from the largest to the smallest request. Since each request has two resources (CPU and memory), the product of the requested CPU and memory is used as a metric when sorting the requests. The intuition behind this sorting criteria is to consider VMs with larger CPU and memory demands for placement first as they are harder to fit since they require larger free space. It is worth mentioning that the product of the requested CPU and memory resources is just used as a sorting metric and the amount of requested CPU and memory resources are actually allocated to each VM request when it is placed on a PM.

The algorithm iterates on the sorted VMs and starts by picking the first VM request in the ordered `VMList`. Next, it iterates over the sorted PMs trying to fit the picked VM within the first PM in the ordered `PMList`. The algorithm keeps iterating over the PMs in `PMList` until the VM request fits one. The PM that fits the picked VM can be either asleep or ON. If it is asleep, then the PM is added to the `ONList` (line 9) and the PM's state is changed to ON (line 10). `ONList` is basically a list that stores the PMs that need to be turned ON to cover the future workload. If the picked PM is not asleep, these two steps are skipped and the algorithm goes directly to line 12. Since the PM will host the picked VM request, the utilization of the PM is updated by calculating the new utilization of the PM after placing the picked VM request (line 12). In line 13, `PMList` is sorted based on the three criteria since at least the utilization (or both the utilization and state) of one of the PMs is changed. The algorithm picks

next the following VM request in `VMList` and tries to fit it similarly in a PM, and repeats the same procedure until all VMs are placed in a PM.

After assigning all of the predicted VMs to a PM, the algorithm checks whether there are PMs that are ON but have zero utilization. If that is the case, then these PMs are added to `SleepList` as they are redundant and need to be switched to the sleep mode (line 20). This case might happen if the cloud clients terminated many VM requests within the last minute and the corresponding PMs that host these VMs became idle. If the predicted workload in the future prediction window can be hosted by a subset of these idle PMs, then the extra PMs are switched to the sleep state to save energy.

The algorithm ends up with two lists: `ONList` and `SleepList` where one of the two lists is empty. It is easy to see that one of the two lists will be empty, as in our PM sorting criteria, machines that are ON come first and thus will be used to host the predicted VMs. More PMs will be turned ON only when all already ON PMs are used; in this case, `SleepList` will be empty as there will be no ON PMs with zero utilization. Finally, the Power Management module turns all the PMs in `ONList` ON or switches all the PMs in `SleepList` to the sleep state and updates the new states in the original `ResourceTable` respectively. As a result, we end up keeping ON only the needed machines to cover the predicted workload, and putting to sleep the rest of machines.

Algorithm 1 Modified BFD – estimating the number of needed PMs during the next prediction window

Input:
ResourceTable: a table whose entries each corresponds to a PM and contains its power state, its CPU and memory capacity and its CPU and memory utilization.
c_1, c_2, ..., c_k: centers of the k categories. Each center is a point in $\mathbb{R}^2$ where the two dimensions are the CPU and memory resources.
$\bar{d}(1)$, $\bar{d}(2)$, ..., $\bar{d}(k)$: predicted numbers of requests for the k categories in the coming prediction window.
Output:
SleepList: List of PMs to be switched to sleep mode.
OnList: List of PMs to be turned on to cover the predicted workload.

```
1: PMTable ← ResourceTable
2: PMTable.sortPMs()
3: VMList ← ConstructList(d̄(1), d̄(2), ..., d̄(k), c1, c2, ..., ck)
4: VMList.sortVMs()
5: for each VM in VMList do
6:    for each PM in PMTable do
7:       if VM fits in PM then
8:          if PM was asleep then
9:             OnList.add(PM)
10:            PMTable.SetState(PM,"ON")
11:         end if
12:         PMTable.UpdateUtilization(PM,VM)
13:         PMTable.sortPMs()
14:         break the for loop and try to pack the next VM
15:      end if
16:   end for
17: end for
18: for each PM in PMList do
19:    if PM.state="ON" AND PM.utilization=0 then
20:       SleepList.add(PM)
21:    end if
22: end for
```

VIII. Framework Evaluation

In this section, we evaluate the performance of our integrated energy-aware cloud resource provisioning framework (with adaptive prediction and safety margin) and compare it against the following schemes[2]:

- **No Power Management:** represents the case where all PMs of the cluster are left ON all the time.
- **PM Prediction Power Management:** this scheme follows the prediction approach that was proposed in [44, 45] which observes the number of PMs that were needed in each minute during the last T_{obs} minutes, predicts the average of those observations to be the number of PMs needed in the coming minute, and scales the number of ON PMs up or down based on this prediction. A fixed number N_{buffer} of extra PMs is left ON as a buffer on top of those predictions to reduce the cases where clients have to wait for a sleeping PM to be switched ON before their requests get allocated. Tuning the parameters of this scheme (T_{obs} and N_{buffer}) is done by selecting the values that achieved the best prediction accuracy on the training data.
- **Optimal Power Management:** represents the case where the predictor knows the exact number of future VM requests, as well as the exact amount of CPU and memory resources associated with these requests. This represents the best-case scenario for resource management and serves here as an upper bound.

A. Resource Utilization

We start first by analyzing how efficient the ON PMs are utilized when the different schemes are used to manage the Google cluster. Figs. 11 and 12 show snapshots that plot the average CPU and memory utilization of the ON PMs over time when evaluating the different schemes on the testing period. For completeness, we also show in Table IV the ON PMs average CPU and memory utilizations averaged over the entire duration of the testing data. Observe that both CPU and memory utilizations are very low when no power management is applied, since many PMs are kept ON without being utilized. The PM Prediction Power Management scheme improves the utilization of the CPU and memory resources compared to the previous scheme, as it observes how many PMs were recently used and predicts broadly how many ON PMs will be needed in future. However, this scheme still wastes resources and is far from the optimal utilization levels. Our framework, on the other hand, improves further the average utilization of both CPU and memory, as it estimates accurately how many ON PMs will be needed by predicting the future requests along with their requested resource demands. In fact, the utilization achieved under our framework is very close to that of the Optimal Power Management scheme. The utilization gap between our framework and the optimal case is mainly due to prediction errors and to the redundant PMs that are left ON as a safety margin. It is worth mentioning that the optimal power management does not achieve 100% utilization due to the bin packing nature, where some PMs may still have some space left (not fully utilized).

[2]The number of ON PMs used by the Google cluster is kept private and thus we could not compare with Google's power management scheme.

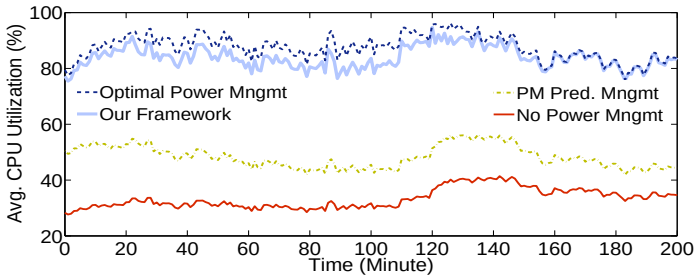

Fig. 11. A comparison of the average CPU utilization.

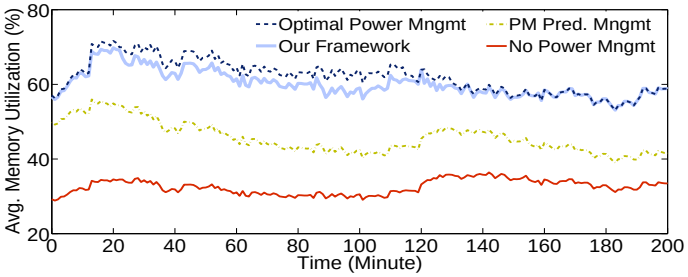

Fig. 12. A comparison of the average memory utilization.

B. Energy Savings

We now assess the total energy consumed by the Google cluster when the traces reported during the testing period are handled by the different resource management schemes. The cluster's total costs include the energy consumed by both ON and sleeping PMs, as well as the transition energy associated with switching a PM from ON to sleep and vice versa. We rely on the power numbers reported in [43] and summarized in Table III to calculate those total costs where the power consumed by a sleeping PM is P_{sleep}, whereas the power consumed by an ON PM increases linearly from P_{idle} to P_{peak} as its CPU utilization increases from 0 to 100%.

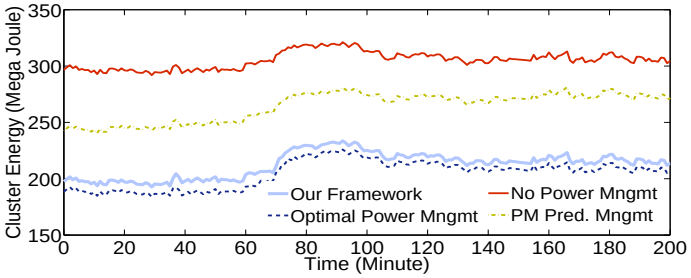

Fig. 13. A comparison of the consumed energy.

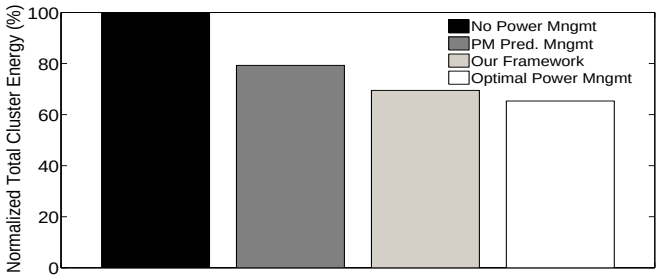

Fig. 14. Total consumed energy during the entire testing period.

TABLE IV
AVG. CPU AND MEMORY UTILIZATION OVER ALL TESTING DATA.

	No Power Mngmt	PM Pred. Mngmt	Our Framework	Opt. Power Mngmt
AvgCPU Util	38.15 %	59.06 %	83.94 %	90.42 %
AvgMem Util	32.69 %	50.37 %	65.34 %	69.92 %

Fig. 13 shows a snapshot of the energy consumed by the cluster when it is handling the requests reported in the testing traces using the different resource management schemes. Observe that by leaving all the PMs ON, the No Power management scheme consumes significant amount of energy over time. By predicting on average how many PMs will be needed using the PM Prediction scheme, the cluster reduces its costs. Whereas by using our proposed framework, the consumed energy is lower than the former two cases as we are decomposing the workload into multiple categories, and predicting the workload for each category which helps in keeping the right amount of needed servers ON. Furthermore, unlike the PM Prediction scheme which leaves a fixed number of ON PMs as a buffer all the time, our framework uses an adaptive safety margin that is proportional to the workload predictions, which avoids keeping too many ON PMs when they will not actually be needed. We also show in Fig. 13 the energy costs of the Optimal Power Management scheme. Results show that the energy costs of our framework are slightly larger than the optimal case. This difference is mainly due to the prediction errors and to the safety margin overhead.

For completeness, Fig. 14 shows the total energy consumed by the Google cluster during the entire testing period for the different schemes normalized with respect to the total energy cost of the No Power Management scheme. The results show how close the energy consumed by our framework is from the optimal case and highlights that a significant portion of the cluster's consumed energy can be saved by estimating accurately the future workload. It is worth mentioning that the energy costs associated with the optimal scheme are inevitable and represent the energy consumed by the hosted workload.

We discussed so far the amount of energy that our framework saves. Now in order to have a sense/estimate of the actual savings in terms of money, observe from Fig. 13 that about 50 Mega Joules are saved by our framework every minute when compared to the PM Prediction scheme. This translates into a total savings of 2160 Giga Joules per month, or equivalently a total of 600 Megawatt-hour per month. Per year, this translates into a saving of 7200 Megawatt-hour. For example, in California in 2013, the commercial cost of one kilowatt-hour is about 17 Cents [46]. Therefore, we can say that our techniques can save the studied Google cluster roughly about $1.2 million per year.

C. Workload Underestimation Evaluation

We assess now how often clients had to wait for a sleeping PM to be switched ON before their requests got allocated. In order to do that, we show in Table V the total number of underestimated PMs that were switched from sleep to the ON state to accommodate the received workload when each of our framework and the PM Prediction Power Management scheme were evaluated on the entire testing data. Clearly, the number of clients that

had to wait before their resources were allocated is proportional to the number of underestimated PMs that had to be awaken from sleep to accommodate the received requests. Observe that our framework had around 65% less underestimation incidents compared to the PM Power Management scheme. This proves the efficiency of our prediction and safety margin techniques compared to the PM Prediction scheme which calculates broadly how many ON PMs will be needed and keeps a fixed number of ON PMs as a buffer to avoid those underestimation cases.

TABLE V
TOTAL NUMBER OF UNDERESTIMATED PMS DURING THE ENTIRE TESTING PERIOD.

	PM Pred. Mngmt	Our Framework
Number of PMs	1270	456

D. Execution Time Overhead Evaluation

Our last experiment analyzes the execution time overhead of the offline as well as the online stages of our framework. All the measurements reported in this section are based on running our framework on a platform that has an Intel Core 2 Quad Q9400 2.66 GHz processor, and an 8 GB RAM.

As for the offline stages, they are executed only once on large training traces in order to capture the general workload characteristics. Recall that our framework has two offline stages: the Clustering stage (used to decide how many categories the requests will be divided to), and the the Predictors Training stage (used to tune initially the parameters for each category's predictor). Table VI shows the total time spent to execute each one of those offline stages on the Google training data.

Online stages on the other hand are performed periodically every minute. We measure how much time each online stage takes each time it is called when running our framework on the Google testing data. Since the time spent by each online stage may slightly vary depending on the workload, we report in Table VI the mean and standard deviation for how much time each online stage takes when it is called by our framework.

The reported measurements in Table VI clearly show that our framework not only achieves great savings but also has a light offline and online execution overhead, which is something important for efficient resource management.

TABLE VI
EXECUTION TIME ANALYSIS OF OUR FRAMEWORK'S STAGES.

Offline Stage	Execution Time	
Clustering	60 Min	
Predictors Training	13 Min	
Online Stage	**Execution Time**	
	Mean	Standard Deviation
Trace Decomposition	0.017 Sec	0.02 Sec
Workload Prediction	0.002 Sec	0.001 Sec
Power Management	2.9 Sec	1 Sec

IX. CONCLUSION AND FUTURE WORK

We propose an integrated energy-aware resource provisioning framework that predicts the number of VM requests and the amount of resources associated with these requests that the cloud center will receive in the future. These predictions are used to keep the right amount of needed PMs ON. The methods to estimate the parameters involved in our framework are presented including setting the number of clustered categories, the length of the prediction window, the optimal weights of the stochastic predictor, and the length of the observation window. The effectiveness of our framework is evaluated using real traces from Google cloud cluster. We show that our framework outperforms existing prediction techniques and achieves significant energy savings and high utilization that are very close to the optimal case. For future work, we plan to investigate whether VM requests follow certain regular daily trends and rely on that to further improve the workload prediction module. Finally, the lack of publicly available fine-grained traces, similar to Google traces, has prevented us from testing our framework on other cloud traces. We hope to have the chance to do that in future.

REFERENCES

[1] R. Brown et al., "Report to congress on server and data center energy efficiency: Public law 109-431," 2008.
[2] "Cisco global cloud index: Frocast and methodology, 2011-2016," *Cicso Inc., White Paper*, 2012.
[3] C. Pettey, "Gartner estimates ICT industry accounts for 2 percent of global co2 emissions," 2007.
[4] T. Taleb, "Toward carrier cloud: Potential, challenges, and solutions," *IEEE Wireless Communications*, vol. 21, no. 3, 2014.
[5] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, 2014.
[6] A. Ksentini, T. Taleb, and F. Messaoudi, "A LISP-based implementation of follow me cloud," *IEEE Access*, vol. 2, 2014.
[7] L. Barroso and U. Holzle, "The case for energy-proportional computing," *Computer Journal*, vol. 40, no. 12, pp. 33–37, 2007.
[8] *http://code.google.com/p/googleclusterdata/*.
[9] H. Hajj, W. El-Hajj, M. Dabbagh, and T.R. Arabi, "An algorithm-centric energy-aware design methodology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2013.
[10] Y. Lu and G. De Micheli, "Comparing system level power management policies," *Design Test of Computers*, vol. 18, no. 2, pp. 10–19, 2001.
[11] B. Kveton, P. Gandhi, G. Theocharous, S. Mannor, B. Rosario, and N. Shah, "Adaptive timeout policies for fast fine-grained power management," in *Proceedings of the National Conference on Artificial Intelligence*, 2007.
[12] A. Weissel and F. Bellosa, "Self-learning hard disk power management for mobile devices," in *Proceedings of the International Workshop on Software Support for Portable Storage (IWSSPS)*, 2006.
[13] S. Albers, "Energy-efficient algorithms," *Communications of the ACM*, vol. 53, no. 5, pp. 86–96, 2010.
[14] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Release-time aware VM placement," in *Proceedings of IEEE GLOBECOM Workshop on Cloud Computing Systems, Networks, and Applications (CCSNA)*, 2014.
[15] Y. Ajiro and A. Tanaka, "Improving packing algorithms for server consolidation," in *Computer Measurement Group (CMG) conference*, 2007.
[16] E. Man Jr, M. Garey, and D. Johnson, "Approximation algorithms for bin packing: A survey," *Jouranl of Approximation Algorithms for NP-Hard Problems*, pp. 46–93, 1996.
[17] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proceedings of IEEE INFOCOM*, 2010.
[18] A. Beloglazov, *Energy-Efficient Management of Virtual Machines in Data Centers for Cloud Computing*, Ph.D. thesis, 2013.
[19] M. Melo, S. Sargento, U. Killat, A. Timm-Giel, and J. Carapinha, "Optimal virtual network embedding: Node-link formulation," *IEEE Transcations on Network and Service Management*, vol. 10, no. 4, pp. 356–368, 2013.
[20] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, 2012.
[21] A. Beloglazov and R. Buyya, "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1366–1379, 2013.
[22] G. Keller, M. Tighe, H. Lutfiyya, and M. Bauer, "A hierarchical, topology-aware approach to dynamic data centre management," in *IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–7.

[23] C. Mastroianni, M. Meo, and G. Papuzzo, "Probabilistic consolidation of virtual machines in self-organizing cloud data centers," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, pp. 215–228, 2013.
[24] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Efficient datacenter resource utilization through cloud resource overcommitment," in *IEEE INFOCOM Workshop on Mobile Cloud and Virtualization*, 2015.
[25] T. Taleb and A. Ksentini, "Follow me cloud: interworking federated clouds and distributed mobile networks," *IEEE Network*, vol. 27, no. 5, 2013.
[26] J. Doyle, R. Shorten, and D. O'Mahony, "Stratus: Load balancing the cloud for carbon emissions control," *IEEE Transactions on Cloud Computing*, vol. 1, no. 1, pp. 1–1, Jan 2013.
[27] Q. Zhang and R. Boutaba, "Dynamic workload management in heterogeneous cloud computing environments," in *IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–7.
[28] S. Garg, C. Yeo, A. Anandasivam, and R. Buyya, "Environment-conscious scheduling of hpc applications on distributed cloud-oriented data centers," *Journal of Parallel and Distributed Computing*, vol. 71, no. 6, 2011.
[29] M. Shojafar, N. Cordeschi, D. Amendola, and E. Baccarelli, "Energy-saving adaptive computing and traffic engineering for real-time-service data centers," in *Proceedings of IEEE ICC Workshop on Cloud Computing Systems, Networks, and Applications*, 2015.
[30] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: elastic distributed resource scaling for infrastructure-as-a-service," in *Proceedings of the USENIX International Conference on Automated Computing*, 2013.
[31] T. Heath, B. Diniz, E. Carrera, W. Meira Jr, and R. Bianchini, "Energy conservation in heterogeneous server clusters," in *Proceedings of ACM SIGPLAN symposium on parallel programming*, 2005.
[32] J. Prevost, K. Nagothu, B. Kelley, and M. Jamshidi, "Prediction of cloud data center networks loads using stochastic and neural models," in *Proceedings of the International Conference on System of Syst. Eng.*, 2011.
[33] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *Proceedings of the International Conference on Network and Service Management (CNSM)*, 2010.
[34] A. Verma, P. Ahuja, and A. Neogi, "pmapper: power and migration cost aware application placement in virtualized systems," in *Middleware*, pp. 243–264. Springer, 2008.
[35] M. Tighe and M. Bauer, "Integrating cloud application autoscaling with dynamic vm allocation," in *IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–9.
[36] C. Dabrowski and F. Hunt, "Using markov chain analysis to study dynamic behaviour in large-scale grid systems," in *Proceedings of the Australasian Symposium on Grid Computing and e-Research-Volume 99*, 2009.
[37] Y. Zhang, W. Sun, and Y. Inoguchi, "CPU load predictions on the computational grid," in *Proceedings of IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2006.
[38] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2011.
[39] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," *Intel Science and Tech. Center for Cloud Computing Technical Report*, 2012.
[40] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "Analysis and lessons from a publicly available google cluster trace," *Technical Report from University of California, Berkeley*, 2010.
[41] A. Mishra, J. Hellerstein, W. Cirne, and C. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, 2010.
[42] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, 2011.
[43] I. Sarji, C. Ghali, A. Chehab, and A. Kayssi, "Cloudese: Energy efficiency model for cloud computing environments," in *Proceedings of IEEE International Conference Energy Aware Computing (ICEAC)*, 2011.
[44] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz, "NapSAC: Design and implementation of a power-proportional web cluster," *ACM SIGCOMM computer communication review*, 2011.
[45] A. Gandhi, M. Harchol-Balter, and M. Kozuch, "Are sleep states effective in data centers?," in *International Green Computing Conference*, 2012.
[46] U.S. Energy Information Administration, *http://www.eia.gov*.

Mehiar Dabbagh received his B.S. degree in Telecommunication Engineering from the University of Aleppo, Syria, in 2010 and the M.S. degree in ECE from the American University of Beirut (AUB), Lebanon, in 2012. During his Master's studies, he worked as a research assistant in Intel-KACST Middle East Energy Efficiency Research Center (MER) at AUB, where he developed techniques for software energy-awareness. Currently, he is a Ph.D. student in ECE at Oregon State University (OSU). His research interests include: Cloud Computing, Network Security and Data Mining.

Bechir Hamdaoui (S'02-M'05-SM'12) is presently an Associate Professor in the School of EECS at Oregon State University. He received the Diploma of Graduate Engineer (1997) from the National School of Engineers at Tunis, Tunisia. He also received M.S. degrees in both ECE (2002) and CS (2004), and the Ph.D. degree in Computer Engineering (2005) all from the University of Wisconsin-Madison. His research interests span various topics in the areas of wireless communications and computer networking systems. He has won the NSF CAREER Award (2009), and is presently an AE for IEEE Transactions on Wireless Communications (2013-present), and Wireless Communications and Computing Journal (2009-present). He also served as an AE for IEEE Transactions on Vehicular Technology (2009-2014) and for Journal of Computer Systems, Networks, and Communications (2007-2009). He served as the program chair for SRC in ACM MobiCom 2011 and many IEEE symposia/workshops, including ICC, IWCMC, and PERCOM. He also served on the TPCs of many conferences, including INFOCOM, ICC, and GLOBECOM. He is a Senior Member of IEEE, IEEE Computer Society, IEEE Communications Society, and IEEE Vehicular Technology Society.

Mohsen Guizani (S'85-M'89-SM'99-F'09) is currently a Professor at the Computer Science & Engineering Department in Qatar University. Qatar. He also served in academic positions at the University of Missouri-Kansas City, University of Colorado-Boulder, Syracuse University and Kuwait University. He received his B.S. (with distinction) and M.S. degrees in EE; M.S. and Ph.D. degrees in CS in 1984, 1986, 1987, and 1990, respectively, all from Syracuse University, Syracuse, New York. His research interests include Wireless Communications, Computer Networks, Cloud Computing, Cyber Security and Smart Grid. He currently serves on the editorial boards of several international technical journals and the Founder and EiC of "Wireless Communications and Mobile Computing" Journal published by John Wiley. He is the author of nine books and more than 400 publications in refereed journals and conferences (with an h-index=30 according to Google Scholar). He received two best research awards. Dr. Guizani is a Fellow of IEEE, member of IEEE Communication Society, and Senior Member of ACM.

Ammar Rayes Ph.D., is a Distinguished Engineer at Cisco Systems and the Founding President of The International Society of Service Innovation Professionals, www.issip.org. He is currently chairing Cisco Services Research Program. His research areas include: Smart Services, Internet of Everything (IoE), Machine-to-Machine, Smart Analytics and IP strategy. He has authored / co-authored over a hundred papers and patents on advances in communications-related technologies, including a book on Network Modeling and Simulation and another one on ATM switching and network design. He is an Editor-in-Chief for "Advances of Internet of Things" Journal and served as an Associate Editor of ACM "Transactions on Internet Technology" and on the Journal of Wireless Communications and Mobile Computing. He received his BS and MS Degrees in EE from the University of Illinois at Urbana and his Doctor of Science degree in EE from Washington University in St. Louis, Missouri, where he received the Outstanding Graduate Student Award in Telecommunications.