Efficient Datacenter Resource Utilization Through Cloud Resource Overcommitment

Mehiar Dabbagh*, Bechir Hamdaoui*, Mohsen Guizani† and Ammar Rayes‡
*Oregon State University, dabbaghm,hamdaoub@onid.orst.edu | †Qatar University, mguizani@ieee.org | ‡Cisco Systems, rayes@cisco.com

***Abstract*—We propose an efficient resource allocation framework for overcommitted clouds that makes great energy savings by 1) minimizing PM overloads via resource usage prediction, and 2) reducing the number of active PMs via efficient VM placement and migration. Using real Google traces collected from a cluster containing more than 12K PMs, we show that our proposed techniques outperform existing ones by minimizing migration overhead, increasing resource utilization, and reducing energy consumption.**

***Index Terms*—Energy efficiency, VM migration, workload prediction, cloud computing.**

I. Introduction

Studies indicate that datacenter servers operate, most of the time, at between 10% and 50% of their maximal utilizations, and that idle/under-utilized servers consume about 50% of their peak power [1]. Therefore, to minimize energy consumption of datacenters, one needs to consolidate cloud workloads into as few servers as possible and switch to sleep the redundant servers.

Upon receiving a client request, the cloud scheduler creates a virtual machine (VM), allocates the requested resources (e.g., CPU and memory) to it, and assigns the VM to one of the cluster's physical machines (PMs). In current cloud resource allocation methods, the amount of resources specified by the client request remains reserved during the whole VM lifetime. A key question arises now: *what percentage of these reserved resources is actually being utilized?* To answer this question, we conduct some measurements on real Google traces [2] and show in Fig. 1 a one-day snapshot of this percentage. Observe that VMs only utilize about 35% and 55% of the requested CPU and memory resources. Our study indicates that CPU and memory resources tend to be overly reserved, leading to substantial resource wastage.

Resource overcommitment [3] is a technique that has been recognized as a potential solution for addressing the above-mentioned wastage issues. It essentially consists of allocating VM resources to PMs in excess of their actual capacities, expecting that these actual capacities will not be exceeded since VMs are not likely to utilize their reserved resources fully. Therefore, it has great potential for saving energy in cloud centers, as VMs can now be hosted on fewer ON PMs.

Resource overcommitment may, however, lead to PM overloading, which occurs when the aggregate of requested resources of the VMs scheduled on some PM does exceed the PM's capacity, potentially resulting in the degradation of the performance of some or all of the VMs running on the overloaded PM. VM migration [4], where some of the VMs hosted by the overloaded PM are moved to other under-utilized or idle PMs, has been adopted as a solution for handling PM overloading. VM migration raises, however, two key challenges, which we address in this paper.

This work was supported in part by Cisco and NSF (NSF CAREER award CNS-0846044).

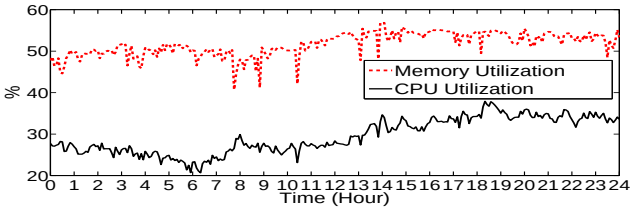

Fig. 1. Resource utilization over one-day snapshot of Google traces.

A. When should VMs be migrated?

Migrations can be triggered after overloads occur [5]. This stops VMs from contending over the limited resources but clearly results in some SLA violations. Another solution is to trigger migrations once the VM's aggregate demands exceeds a certain threshold, but before overloads occur [6–8]. Such techniques have the limitation of triggering many unnecessary migrations. To tackle this, we dedicate in our proposed framework a module that predicts the VM's future resource demands and triggers VM migrations once an overload is foreseen. Our module differs from the previously proposed offline predictive-based techniques [9] in that it learns and predicts the resource demands of VMs *online* without requiring any aprior knowledge about the hosted VMs.

B. Which VMs should be migrated and which PMs these VMs should be migrated to?

Largest First heuristics [10, 11] migrates VMs with the largest resource usages to the PMs with the largest slacks. While these techniques minimize the total number of migrations, they ignore the migration energy overhead. VM moving costs are accounted for in [6]. Although this heuristic tends to move VMs with the lowest moving costs, it is not always guaranteed that there are already-ON PMs with enough slack to host the migrated VMs, forcing turning some PMs from sleep to ON, which comes with a high energy cost [12]. Unlike previous work, our proposed framework has an energy-aware migration module that makes migration decisions in a way that minimizes the total migration energy overhead which is made up of the energy spent to move VMs and that of switching PMs ON to host the migrated VMs.

To recap, we propose in this paper an integrated energy-aware resource allocation framework for overcommited clouds that:

- predicts future resource utilizations of scheduled VMs online, and uses these predictions to make efficient cloud resource overcommitment decisions to increase utilization.
- predicts PM overload incidents before occurring and triggers VM migrations to avoid SLA violations.
- performs energy-efficient VM migration by determining which VMs to migrate and which PMs need to host the migrated VMs to reduce the number of active PMs.

The effectiveness of our techniques is evaluated by real traces collected from a Google cluster containing more than 12K PMs.

The remainder is organized as follows. Section II briefly describes the component of our proposed framework. Section III presents our proposed prediction methods. Section IV formulates the VM migration as an optimization problem, and section V presents a heuristic for solving it. Section VI presents our evaluation results. Finally, Section VII concludes the paper.

II. Proposed Framework

Our proposed framework is suited for heterogeneous cloud clusters whose PMs may or may not have different resource capacities. We consider in this work two cloud resources: CPU and memory, although our framework can easily be extended to any number of resources. Thus, a PM j can be represented by $[C^j_{cpu}, C^j_{mem}]$, where C^j_{cpu} and C^j_{mem} are the PM's CPU and memory capacities. Throughout, let P be the set of all PMs in the cloud cluster, and $\vec{C}_{cpu} = (C^1_{cpu}, C^2_{cpu}, \ldots, C^{|P|}_{cpu})$ and $\vec{C}_{mem} = (C^1_{mem}, C^2_{mem}, \ldots, C^{|P|}_{mem})$. Recall that a client may, at any time, submit a new VM request, say VM i, represented by $[R^i_{cpu}, R^i_{mem}]$ where R^i_{cpu} and R^i_{mem} are the requested amounts of CPU and memory. Whenever the client no longer needs the requested resources, it submits a VM release request. Throughout, let V be the set of all VMs hosted by the cluster.

We briefly describe next our framework components so as the reader will have a global picture of the entire framework before delving into the details. Throughout this section, Fig. 2 is used for illustration. Our framework is made up of five modules:

A. VM Utilization Predictor

This module predicts the resource utilizations of all of the already admitted VMs. For each scheduled VM i, two predictors (one for CPU and one for memory) monitor and collect the VM's CPU and memory usage traces, and use them, along with other VM parameter sets (to be learned online from the VM's resource utilization behaviors), to predict the VM's future CPU and memory utilizations, P^i_{cpu} and P^i_{mem}. Throughout, these parameter sets will be denoted for VM i by $Param(cpu, i)$ and $Param(mem, i)$ for CPU and memory, respectively. The CPU and memory predictions for all VMs, $\vec{P}_{cpu} = (P^1_{cpu}, P^2_{cpu}, \ldots, P^{|V|}_{cpu})$ and $\vec{P}_{mem} = (P^1_{mem}, P^2_{mem}, \ldots, P^{|V|}_{mem})$, are then passed as an input to the following module. These predictions are calculated for the coming τ period and are done periodically at the end of each period. Detailed description of how predictors work and how these parameters are updated are given in section III.

B. PM Aggregator

This module takes as an input the VMs' predicted CPU and memory utilizations, $\vec{P}_{cpu}$ and $\vec{P}_{mem}$, and returns the PMs' predicted aggregate CPU and memory utilizations, $\vec{U}_{cpu} = (U^1_{cpu}, U^2_{cpu}, \ldots, U^{|P|}_{cpu})$ and $\vec{U}_{mem} = (U^1_{mem}, U^2_{mem}, \ldots, U^{|P|}_{mem})$, where U^j_{cpu} and U^j_{mem} are calculated for each PM j as

$$U^j_{cpu} = \sum_{i \in V : \theta(i) = j} P^i_{cpu} \text{ and } U^j_{mem} = \sum_{i \in V : \theta(i) = j} P^i_{mem}$$

where $\theta : V \rightarrow P$ is the VM-PM mapping function, with $\theta(i) = j$ meaning that VM i is hosted on PM j.

C. PM Overload Predictor

This module monitors and predicts overloads before they occur. It takes as input $\vec{U}_{cpu}$ and $\vec{U}_{mem}$ along with PMs' capacities, $\vec{C}_{cpu}$ and $\vec{C}_{mem}$, and returns O_{pm}, the set of PMs that are predicted to overload. That is, a PM j is added to O_{pm} when $U^j_{cpu} > C^j_{cpu}$ or $U^j_{mem} > C^j_{mem}$. As expected with any prediction framework, it is also possible that our predictors fail to predict an overload. We refer to such incidents as *unpredicted overloads*, which will be eventually detected when they occur. For any predicted PM overload, VM migration will be performed before the overload actually occurs, thus avoiding it. But for each unpredicted PM overload, VM migration will be performed upon its detection. All VM migrations are handled by the next module.

D. Energy-Aware VM Migration

This module determines which VM(s) among those hosted on the PMs in O_{pm} need to be migrated so as to keep the predicted aggregate CPU and memory utilizations below the PM's capacity. To make efficient decisions, the module needs to know the energy costs for moving each scheduled VM, referred to by $\vec{m} = \{m_1, m_2, ..., m_{|V|}\}$ where m_i is the cost (in Joules) for moving the i^{th} VM. This module also determines which PM each migrating VM needs to migrate to. Such a PM must have enough CPU and memory slack to accommodate the migrated VM(s), and thus the module needs to know the PMs' capacities (i.e., $\vec{C}_{cpu}$ and $\vec{C}_{mem}$), as well as the PMs' predicted aggregate utilizations (i.e., $\vec{U}_{cpu}$ and $\vec{U}_{mem}$). These are provided to the module as input in addition to the ON-sleep states of the PMs γ, where the state function $\gamma(j)$ returns PM j's power state (ON or sleep) prior to migration. The output of this module is the new VM-PM mapping θ and the new ON-sleep PM state γ after all VM migrations are taken place. Details on how the the migration problem is formulated as an optimization problem and how it is solved by our module are provided in sections IV and V.

E. PM Allocation

This module decides where to place newly submitted VMs and also handles VM release events. The new VM placement are handled with two objectives in mind: saving energy and minimizing PM overload occurrence probability. In order to do so, the predicted CPU and memory slacks, S^j_{cpu} and S^j_{mem}, are first calculated for each PM $j \in P$ as:

$$S^j_{cpu} = C^j_{cpu} - U^j_{cpu} \text{ and } S^j_{mem} = C^j_{mem} - U^j_{mem} \qquad (1)$$

Then, the PM allocation module sorts PMs based on the following criteria (in ascending order):

(i) PMs that are ON
(ii) PMs with the largest *predicted slack metric* which is defined for a PM j as $S^j_{cpu \times mem} = S^j_{cpu} \times S^j_{mem}$.

The intuition behind our sorting criteria is as follows: It is better to host a newly submitted VM request on an ON PM, so as to avoid wakening up asleep machines. This saves energy. On the other hand, our predictions imply that the PM with the largest slack is less likely to experience an overload. Hence, it is desirable to pick the PM with the largest predicted slack so as to decrease the chances of overloading PMs.

Once PMs are sorted as above, we then check each PM in the ordered list to see whether it has enough CPU and memory

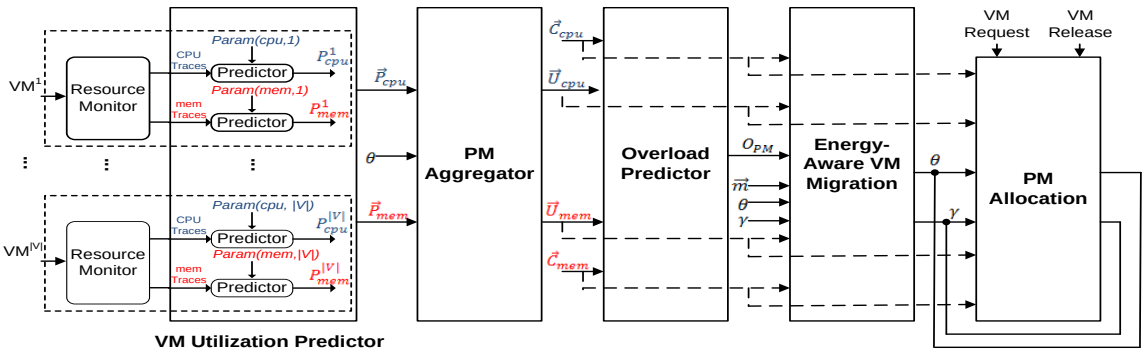

Fig. 2. Flow chart of the proposed framework

resource slack to host the newly submitted VM, and if one PM is found, we schedule the new VM on it. If the found PM happens to be in the sleep state, it is turned ON to host the VM. For this newly scheduled VM, two predictors (one for CPU and one for memory) are then built to monitor the resource utilization behaviors of the VM, as described earlier.

When a client no longer needs its VM, it submits a VM release. Upon receiving a VM release, the PM allocation module releases the VM's allocated CPU and memory resources, frees all system parameters associated with the VM (e.g., predictors), and updates the aggregate CPU and memory predictions of the hosting PM accordingly. The PM is switched to the sleep state if it becomes idle after releasing the VM.

III. VM Utilization Predictor

We explain in this section how a predictor for a scheduled VM predicts its future resource demands in the coming τ minutes, where the term resource will be used to refer to either the CPU or memory. In our framework, we choose to use the Wiener filter prediction approach for several reasons. First, it is simple and intuitive, as the predicted utilization is a weighted sum of the recently observed utilization samples. Second, it has a sound theoretical basis. Third, weights can easily be updated without requiring heavy calculations or large storage space. Finally, it performs well on real traces as will be seen later.

Let n be the time at which resource predictions for a VM need to be made. We use the following notations:

- $z[n-i]$: is the VM's average resource utilization during period $[n-(i+1)\tau, n-i\tau]$ minutes.
- $d[n]$: is the VM's actual average resource utilization during period $[n, n+\tau]$.
- $\hat{d}[n]$: is the VM's predicted average resource utilization during period $[n, n+\tau]$.

Wiener filters predict resource utilizations while assuming wide-sense stationarity of $z[n]$. The predicted average resource utilization, $\hat{d}[n]$, is a weighted average over the L most recent observed utilization samples; i.e., $\hat{d}[n] = \sum_{i=0}^{L-1} w_i z[n-i]$, where w_i is the i^{th} sample weight. The prediction error, $e[n]$, is then the difference between the actual and predicted utilizations; i.e., $e[n] = d[n] - \hat{d}[n] = d[n] - \sum_{i=0}^{L-1} w_i z[n-i]$. The objective is to find the weights that minimize the Mean Squared Error (MSE) of the training data, where $MSE = E[\,e^2[n]\,]$. Differentiating MSE with respect to w_k and setting this derivative to zero yields, after some algebraic simplifications, $E\big[d[n]z[n-k]\big] - \sum_{i=0}^{L-1} w_i E\big[z[n-k]z[n-i]\big] = 0$. It then follows that $r_{dz}(k) = \sum_{i=0}^{L-1} w_i r_{zz}(i-k)$ where

$$r_{dz}(k) = E\big[d[n]z[n-k]\big] \tag{2}$$

$$r_{zz}(i-k) = E\big[z[n-k]z[n-i]\big] \tag{3}$$

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

$$R_{zz} = \begin{bmatrix} r_{zz}(0) & r_{zz}(1) & \dots & r_{zz}(L-1) \\ r_{zz}(1) & r_{zz}(0) & \dots & r_{zz}(L-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_{zz}(L-1) & r_{zz}(L-2) & \dots & r_{zz}(0) \end{bmatrix}$$

$$W = \begin{bmatrix} w_0 & w_1 & \dots & w_{L-1} \end{bmatrix}^T$$

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

Given R_{zz} and R_{dz}, the weights can then be calculated as:

$$W = R_{zz}^{-1} R_{dz} \tag{4}$$

The elements of R_{zz} are calculated using the unbiased correlation estimation as:

$$r_{zz}(i) = \frac{1}{N-i} \sum_{j=0}^{N-i-1} z[j+i]z[j] \tag{5}$$

where N is the VM's number of observed samples with each sample representing an average utilization over τ minutes.

The elements of R_{dz} can also be estimated using the correlation coefficients. Since $d[n]$ represents the average resource utilization in the coming τ minutes, we can write $d[n] = z[n+1]$. Plugging the expression of $d[n]$ in Eq. (2) yields $r_{dz}(k) = E[z[n+1]z[n-k]] = r_{zz}(k+1)$, and thus $R_{dz} = \begin{bmatrix} r_{zz}(1) & r_{zz}(2) & \dots & r_{zz}(L) \end{bmatrix}^T$. The elements of R_{dz} can be calculated using Eq. (5). An MSE estimation of the weight vector follows then provided R_{dz} and R_{zz}.

Recall that each VM requests two resources: CPU and memory. Hence, two predictions $\hat{d}^v_{cpu}[n]$ and $\hat{d}^v_{mem}[n]$ are calculated as described above for each VM v. Since the resource utilizations can't exceed the requested capacity specified by the client then our final resource predictions for VM v are: $P^v_{cpu} = \min(\hat{d}^v_{cpu}[n], R^v_{cpu})$ and $P^v_{mem} = \min(\hat{d}^v_{mem}[n], R^v_{mem})$.

Note that $N = L+1$ samples need to be observed in order to calculate the predictor's weights. When the number of samples

available during the early period is less than $L+1$, no prediction will be made and we assume that VMs will utilize all of their requested resources. Once the predictor observes $N = L + 1$ samples, the correlations $r_{zz}(i)$ can then be calculated for all i allowing the weights to be estimated.

When $N > L + 1$, the predictor adapts to new changes by observing the new utilization samples, updates the correlations, and calculates the new updated weights. This results in increasing the accuracy of the predictor over time, as the weights are to be calculated based on a larger training data. From Eq. (5), the coefficient $r_{zz}(i)$ can be written as $Sum(i)/Counter(i)$, where $Sum(i) = \sum_{j=0}^{N-i-1} z[j+i]z[j]$ and $Counter(i) = N-i$ are two aggregate variables. Now recall that every τ minutes, a new resource utilization sample $z[k]$ is observed, and hence, the aggregate variables can be updated as $Sum(i) \leftarrow Sum(i)+z[k]z[k-i]$ and $Counter(i) \leftarrow Counter(i)+1$ and the correlation $r_{zz}(i)$ is updated again as $Sum(i)/Counter(i)$. The updated weights are then calculated using Eq. (4), which will be used to predict the VM's future resource utilizations. Note that only two variables need to be stored to calculate r_{zz} instead of storing all the previous traces, and thus the amount of storage needed to update these weights is reduced significantly.

IV. Energy-Aware VM Migration

VM migration must be performed when an overload is predicted in order to avoid SLA violations. Since energy consumption is our primal concern, we formulate the problem of deciding which VMs to migrate and which PMs to migrate to as an optimization problem with the objective of minimizing the migration energy overhead as described next.

Decision Variables. Let O_{vm} be the set of VMs that are currently hosted on all the PMs that are predicted to overload in O_{pm}. For each VM $i \in O_{vm}$ and each PM $j \in P$, we define a binary decision variable x_{ij} where $x_{ij} = 1$ if VM i is assigned to PM j after migration, and $x_{ij} = 0$ otherwise. Also, for each $j \in P$, we define $y_j = 1$ if at least one VM is assigned to PM j after migration, and $y_j = 0$ otherwise.

Objective Function. Our objective is to minimize VM migration energy overhead, which can be expressed as

$$\sum_{i \in O_{vm}} \sum_{j \in P} x_{ij}\, a_{ij} + \sum_{j \in P} y_j\, b_j \tag{6}$$

and is constituted of two components: *VM moving energy overhead* and *PM switching energy overhead*. VM moving energy overhead, captured by the left-hand summation term, represents the energy costs (in Joule) associated with moving VMs from overloaded PMs. The constant a_{ij} represents VM i's moving cost, and is equal to m_i when VM i is moved to a PM j different from its current PM, and equal to 0 when VM i is left on the same PM where it has already been hosted. Formally, $a_{ij} = 0$ if, before migration, $\theta(i) = j$, and $a_{ij} = m_i$ otherwise. Here m_i denotes VM i's moving energy overhead.

PM switching energy overhead, captured by the right-hand term of the objective function (Eq. (6)), represents the energy cost associated with switching PMs from sleep to ON to host the migrated VMs. The constant $b_j = 0$ if PM j has already been ON before migration (i.e., $\gamma(j)$ = ON before migration), and $b_j = E_{s \to o}$ otherwise, where $E_{s \to o}$ is the transition energy consumed when switching a PM from sleep to ON.

Constraints: The optimization problem is subject to the following constraints. One,

$$\sum_{j \in P} x_{ij} = 1 \quad \forall i \in O_{vm}$$

dictating that every VM must be assigned to only one PM. Two,

$$\sum_{i \in O_{vm}} x_{ij}\, P^i_{cpu} \le C^j_{cpu} \quad \forall j \in O_{pm}$$

$$\sum_{i \in O_{vm}} x_{ij}\, P^i_{mem} \le C^j_{mem} \quad \forall j \in O_{pm}$$

which state that the predicted CPU and memory usage of the scheduled VMs on any overloaded PM must not exceed the PM's available CPU and memory capacities. Three,

$$\sum_{i \in O_{vm}} x_{ij}\, P^i_{cpu} \le S^j_{cpu} \quad \forall j \in P \backslash O_{pm}$$

$$\sum_{i \in O_{vm}} x_{ij}\, P^i_{mem} \le S^j_{mem} \quad \forall j \in P \backslash O_{pm}$$

where $P \backslash O_{pm}$ is the set of PMs predicted not to be overloaded. S^j_{cpu} and S^j_{mem} are the predicted CPU and memory slacks for PM j calculated using Eq.(1). Recall that some VMs will be migrated to PMs that already have some scheduled VMs, and the above constraints ensure that there will be enough resource slack to host any of the migrated VMs. Four,

$$\sum_{i \in O_{vm}} x_{ij} \le |O_{vm}|\, y_j \quad \forall j \in P \tag{7}$$

which forces y_j to be one (i.e., PM j needs to be ON) if one or more VMs in O_{vm} will be assigned to PM j after migration.

Note that if none of the VMs in O_{vm} is assigned to PM j, then the constraint (7) can still hold even when y_j takes on the value one. In order to force y_j to be zero when no VM is assigned to PM j (i.e. PM j maintains the same power state that it had prior to migration as no VM will be migrated to it), we add the following constraint. Five,

$$1 + \sum_{i \in O_{vm}} x_{ij} > y_j \quad \forall j \in P \tag{8}$$

Note that if one or more VMs is assigned to PM j, constraint (8) does not force $y_j = 1$ either, but constraint (7) does. Thus, constraints (7) and (8), together, imply that $y_j = 1$ if and only if one or more VMs are assigned to PM j after migration.

After solving the above problem, the optimal y_js indicate whether new PMs need to be turned ON (also reflected via the γ function), and the optimal x_{ij}s indicate whether new VM-PM mappings are needed (also reflected via the θ function).

V. Proposed Heuristic

In the previous section, we formulated the VM migration problem as an integer linear program (ILP). The limitation of this formulation lies in its complexity, arising from the integer variables, as well as the large numbers of PMs and VMs. To overcome this complexity, we instead propose to solve this problem using the following proposed fast heuristic.

Instead of deciding where to place the VMs that are currently hosted on all the overloaded PMs, our proposed heuristic (shown in Algorithm 1) takes only one overloaded PM P_o at a time

(line 1), and solves a smaller optimization problem to decide where to place the VMs that are currently hosted on the picked PM P_o. We refer to these VMs that are hosted on P_o prior to migration by O_s (line 2). Another feature adopted by our heuristic that reduces the complexity further is to consider only a set of N_{on} ON PMs and N_{sleep} asleep PMs as destination candidates for VM migrations. The set of selected ON PMs, denoted by P_{on}, is formed by the function `PickOnPMs` in line 4. The returned PMs by this function are the ON PMs that have the largest predicted slack metric ($S_{cpu} \times S_{mem}$). The intuition here is that the PM with the largest predicted slack has higher chances for having enough space to host the VMs that need to be migrated. Furthermore, moving VMs to these PMs has the lowest probability to trigger an overload on these PMs. The set of selected PMs that are asleep is denoted by P_{sleep} and formed using the function `PickSleepPMs`. The returned PMs are the ones that are asleep and that have the largest capacity ($C_{cpu} \times C_{mem}$). Again the intuition here is that PMs with larger capacity have larger space to host the migrated VMs and hence, the lowest probability of causing an overload.

The heuristic then forms the set P_s (line 5) which is made up of the picked overloaded PM, P_o, the selected ON PMs, P_{on}, and the selected sleep PMs, P_{sleep}. An optimization problem similar to the one explained in Section IV is solved with the only exception that $P = P_s$ and $O_{vm} = O_s$. Solving the optimization problem determines which VMs in O_s need to be migrated to avoid the PM overload. The VMs that are assigned to one of the ON PMs are then migrated to the assigned ON PMs. As for the VMs that are assigned to one of the PMs that are in sleep, we try first to place them in any already ON PMs. To do so, all the ON PMs apart from those in P_{on} are ordered in a decreasing order of their slacks. The VMs that are assigned to the PMs in sleep are also ordered from largest to smallest. We iterate over these VMs while trying to fit them in one of the ordered ON PMs. This is done in order to make sure that no ON PMs (other than the selected PMs in N_{on}) have enough space to host the migrated VMs. If an already ON PM has enough space to host one of these VMs, then the VM is migrated to the ON PM rather than to the PM in sleep so as to avoid incurring switching energy overhead. Otherwise, the VMs are migrated to the assigned PMs that are asleep, as indicated in line 6.

As for N_{on} and N_{sleep}, these parameters affect the size of the optimization problem. The larger the values of these parameters, the higher the number of PM destination candidates, and the longer the time needed to solve the problem. Our experimental results reported in the following section show that for small values of $N_{on} = 1$ and $N_{sleep} = 1$, the problem can be solved very quickly while achieving significant energy savings.

Algorithm 1 [θ, γ] = Proposed Heuristic(N_{on}, N_{sleep})

1: **for** each $P_o \in O_{pm}$ **do**
2: $O_s = \{\forall i \in V \, s.t. \, \theta(i) = P_o\}$
3: $P_{on} \leftarrow$ PickOnPMs(N_{on})
4: $P_{sleep} \leftarrow$ PickSleepPMs(N_{sleep})
5: $P_s \leftarrow P_{on} \cup P_{sleep} \cup P_o$
6: $[\vec{x}, \vec{y}] = SolveOptimization(P_s, O_s)$
7: Migrate VMs that should be placed on a PM $\in P_{on}$
8: Try placing VMs that should be placed on a PM $\in P_{sleep}$ on any ON PM $\notin P_{on}$
9: Update θ and γ
10: **end for**

VI. Framework Evaluation

The experiments presented in this section are based on real traces of the VM requests submitted to a Google cluster that is made up of more than 12K PMs (see [2] for further details). Since the size of the traces is huge, we limit our analysis to a chunk spanning a 24-hour period. Since the traces do not reveal the energy costs associated with moving the submitted VMs ($\vec{m}$ in Fig. 2), we assume that the VM's moving overhead follows a Gaussian distribution with a mean $\mu = 350\,Joule$ and a standard deviation $\delta = 100$. The selection of these numbers is based on the energy measurements reported in [4], which show that the moving overhead varies between 150 and 550 Joules for VM sizes between 250 and 1000 Mega Bytes. These moving costs include the energy consumed by the source PM, the destination PM, and the network links.

As for the power consumed by an active PM, $P(\eta)$, it increases linearly from P_{idle} to P_{peak} as its CPU utilization, $\eta = U_{cpu}/C_{cpu}$, increases from 0 to 100% [1]. More specifically, $P(\eta) = P_{idle} + \eta(P_{peak} - P_{idle})$, where $P_{peak} = 300$ and $P_{idle} = 150$ Watts. A sleeping PM, on the other hand, consumes $P_{sleep} = 100$ Watts. The energy consumed when switching a PM from sleep to ON is $E_{s \to o} = 4260$ Joules, and that when switching a PM from ON to sleep is $E_{o \to s} = 5510$ Joules. These numbers are based on real servers' specs [13].

Overload Prediction. We start our evaluations by showing in Fig. 3 the number of overloads predicted when our framework is run over the 24-hour trace period. These overload predictions are based on predicting the VM's CPU and memory demands in the coming $\tau = 5$ minutes. Although our framework works for any τ value, the selection of $\tau = 5$ is based on the fact that Google traces report resource utilization for the scheduled VMs every 5 minutes. The number of the most recent observed samples considered in prediction is $L = 6$ as our experiments showed that considering more samples would increase the calculation overhead while making negligible accuracy improvements. For the sake of comparison, we also show in Fig. 3 the number of overloads that were not predicted by our predictors. Observe how low the number of unpredicted overloads is; it is actually zero with the exception of three short peaks. This proves the effectiveness of our framework vis-a-vis of predicting overloads ahead of time, thus avoiding VM performance degradation.

VM Migration Energy Overhead. We plot in Fig. 4 the total migration energy overhead (including both VM moving and PM switching overheads) incurred by the migration decisions of our proposed heuristic to avoid/handle the overloads reported in Fig. 3 along with the total energy overhead associated with the migration decisions of the two existing heuristics: Largest First [10, 11] and Sandpiper [6]. Both of these heuristics handle multi-dimensional resources by considering the product metrics, and both select the PM with the largest slack as a destination for each migrated VM.

Observe in Fig. 4 that our proposed heuristic incurs significantly lesser overhead when compared to the other two heuristics. This is attributed to the fact that unlike previous approaches, our heuristic takes both the VM moving overhead and the PM switching overhead when making migration decisions leading into lower total migration energy overhead.

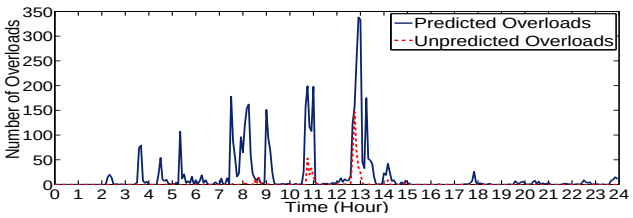

Fig. 3. Number of predicted and unpredicted overloads over time.

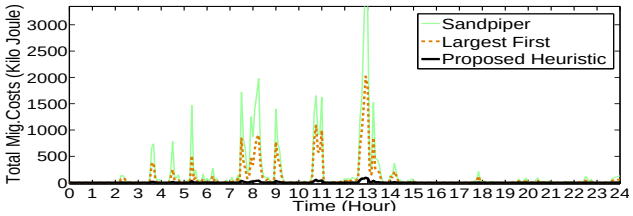

Fig. 4. Total migration energy overhead under each of the three heuristics.

Number of Active PMs. Since the energy consumed by ON PMs constitutes a significant amount, we analyze in Fig. 5 the number of ON PMs when running our framework on the Google traces under each of the three studied migration heuristics. Recall that each migration heuristic makes different decisions to handle PM overloads, and these decisions affect the number of ON PMs, as new PMs may be switched ON to accommodate the migrated VMs. We also show the number of ON PMs when no overcommitment is applied. This represents the case when the exact amount of requested resources is allocated for each VM during its entire lifetime. By comparing these results, observe that after a couple of learning hours, our proposed prediction framework leads to smaller numbers of ON PMs when compared with the case of no overcommittment, and this is true regardless of the VM migration heuristic being used. Also, observe that our proposed prediction techniques, when coupled with our proposed VM migration heuristic, leads to the smallest number of ON PMs when compared with Largest First and Sandpipper heuristics, resulting in greater energy savings. It is worth mentioning that during the first couple of hours, the number of ON PMs is the same regardless of whether resource overcommitment is employed and regardless of the migration technique being used, simply because prediction can't be beneficial at the early stage, as some time is needed to learn from past traces to be able to make good prediction about VMs' future utilizations.

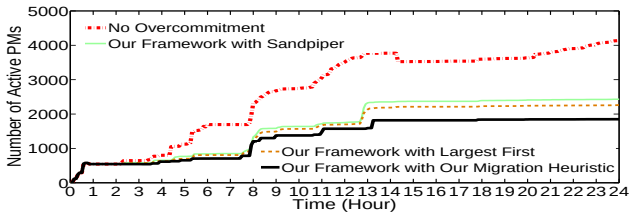

Fig. 5. Number of PMs needed to host the workload.

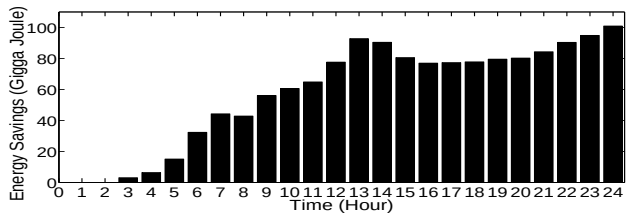

Fig. 6. Energy savings our resource allocation framework achieves when compared to allocation without overcommitment.

Energy Savings. Finally, Fig. 6 shows the total energy savings when the Google cluster adapts our integrated framework (the proposed prediction approach and the proposed migration heuristic) compared to no overcommitment. Observe that savings are not substantial at the beginning as the prediction module needs some time to learn the resource demands of the hosted VMs, but these savings quickly increase over time as the predictors start to observe larger traces and tune their parameters more accurately. Finally, it is clear from Fig 6 that although our framework incurs migration energy overheads (due to both VM moving and PM switching energy overheads) that would not otherwise be present when no overcommitment is applied, the amount of energy saved due to the reduction of the number of ON PMs is much higher than the amount of energy incurred due to migration energy, leading, at the end, to greater energy savings.

VII. Conclusion

We propose an integrated energy-efficient, prediction-based VM placement and migration framework for cloud resource allocation with overcommitment. We show that our proposed framework reduces the number of PMs needed to be ON and decreases migration overheads, thereby making significant energy savings. All of our findings are supported by evaluations conducted on real traces from a Google cluster.

References

[1] L. Barroso and U. Holzle, "The case for energy-proportional computing," *Computer Journal*, vol. 40, pp. 33–37, 2007.

[2] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, 2011.

[3] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Towards energy-efficient cloud computing: Prediction, consolidation, and overcommitment," *IEEE Network Magazine*, 2015.

[4] H. Liu, C. Xu, H. Jin, J. Gong, and X. Liao, "Performance and energy modeling for live migration of virtual machines," in *international symposium on High performance distributed computing*, 2011.

[5] X. Wang and Y. Wang, "Coordinating power control and performance management for virtualized server clusters," *IEEE Transactions on Parallel Distributed Systems*, vol. 22, no. 2, pp. 245–259, 2011.

[6] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Computer Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.

[7] "Vmware distributed power management concepts and use," *VMware Inc., White Paper*, 2010.

[8] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency and Computation: Practice and Experience*, 2012.

[9] L. Chen and H. Shen, "Consolidating complementary VMs with spatial/temporal-awareness in cloud datacenters," in *Proceedings of IEEE INFOCOM*, 2014, pp. 1033–1041.

[10] M. Andreolini, S. Casolari, M. Colajanni, and M. Messori, "Dynamic load management of virtual machines in cloud architectures," in *Cloud Computing*, pp. 201–214. Springer, 2010.

[11] X. Zhang, Z. Shae, S. Zheng, and H. Jamjoom, "Virtual machine migration in an over-committed cloud," in *Network Operations and Management Symposium (NOMS)*, 2012.

[12] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Energy-efficient cloud resource management," in *Proceedings of IEEE INFOCOM Workshop on Moblie Cloud Computing*, 2014.

[13] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Release-time aware VM placement," in *Proceedings of IEEE Globecom*, 2014.