A short version of this paper appears in the 7th International Conference on Network and Service Management, Paris, France, 24-28 October, 2011.

Gossip-based Resource Allocation for Green Computing in Large Clouds (long version)

July 27, 2011

Rerngvit Yanggratoke, Fetahi Wuhib and Rolf Stadler
ACCESS Linnaeus Center
KTH Royal Institute of Technology
Email: {rerngvit,fetahi,stadler}@kth.se

***Abstract*—We address the problem of resource allocation in a large-scale cloud environment, which we formalize as that of dynamically optimizing a cloud configuration for green computing objectives under CPU and memory constraints. We propose a generic gossip protocol for resource allocation which can be instantiated for specific objectives. We develop an instantiation of this generic protocol which aims at minimizing power consumption through server consolidation, while satisfying a changing load pattern. This protocol, called GRMP-Q, provides an efficient heuristic solution that performs well in most cases—in special cases it is optimal. Under overload, the protocol gives a fair allocation of CPU resources to clients. Simulation results suggest that key performance metrics do not change with increasing system size, making the resource allocation process scalable to well above 100,000 servers. Generally, the effectiveness of the protocol in achieving its objective increases with increasing memory capacity in the servers.**

***Index Terms*—cloud computing, green computing, distributed management, power management, resource allocation, gossip protocols, server consolidation**

I. Introduction

Power consumption in datacenters is significant; it has been growing rapidly in recent years, and this growth is expected to continue, as several studies show [1]–[3]. An effective approach to reducing the power consumption of datacenters is *server consolidation* [4], [5], which aims at concentrating the workload onto a minimal number of servers. It is effective, because utilization levels in datacenters today are often low, around 15% [6]. As a running server consumes upwards of 60% of its maximum power consumption, even if it does not carry load (cf. [4]), switching servers that are (temporarily) not needed to a mode that requires minimal or zero power can significantly reduce power consumption.

All key enabling technologies required for server consolidation are available today. *Virtualization* and *live migration* technologies support dynamic consolidation of workload under changing demand. Having various levels of *standby modes* (characterized by different levels of power consumption and wakeup time) that modern equipment offers allows to adapt datacenter resources to changing needs.

In this work, we address the problem of resource management for a large-scale cloud environment (ranging to above 100,000 servers) with the objective of serving a dynamic workload with minimal power consumption. While our contribution is relevant in a more general context, we conduct the discussion from the perspective of the Platform-as-a-Service (PaaS) concept, with the specific use case of a cloud service provider which hosts sites in a cloud environment. The stakeholders in this use case are depicted in figure 1a. The cloud service provider owns and administers the physical infrastructure, on which cloud services are provided. It offers hosting services to site owners through a middleware that executes on its infrastructure (see figure 1b). Site owners provide services to their respective users via sites that are hosted by the cloud service provider. An alternative perspective for discussing our contribution could be based on the Infrastructure-as-a-Service (IaaS) concept. A use case for this concept would include a cloud tenant with a collection of virtual appliances that are hosted on the cloud infrastructure, with services provided to end users through the public Internet. For both use cases, this paper introduces a resource allocation protocol that dynamically places site modules (or virtual machines running virtual appliances respectively) on servers within the cloud, following global management objectives.

The results from this work contribute to engineering a middleware layer that performs resource allocation in

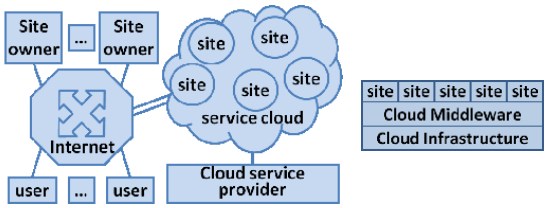


Fig. 1. (a) Deployment scenario with the stakeholders of the cloud environment considered in this work. (b) Overall architecture of the cloud environment; this work focuses on resource management performed by the middleware layer [7].

a cloud environment, with the following design goals.

1) Performance objective: (a) When the system is in underload, the objective is to minimize power consumption through server consolidation while satisfying the demand of hosted sites; (b) When the system is in overload, the objective is to allocate the available resources fairly across hosted sites.
2) Adaptability: The resource allocation process must dynamically and efficiently adapt to changes in the demand.
3) Scalability: The resource allocation process must be scalable both in the number of servers in the cloud and the number of sites the cloud hosts. Specifically, the resources consumed per server in order to achieve a given performance objective must increase sublinearly with both the number of servers and the number of sites.

Our approach centers around a decentralized design whereby the components of the middleware layer run on every server of the cloud environment. (We refer to a server of the cloud as a *machine* in the remainder of this paper.) To achieve scalability, we envision that all key tasks of the middleware layer, including estimating global states, placing site modules and computing policies for request forwarding are based on distributed algorithms.

How do the solutions presented in this paper relate to available management software for private clouds, such as OpenNebula [8], Eucalyptus [9], etc. (IaaS solutions) or AppScale [10], Cloud Foundry [11], etc. (PaaS solutions) All these software packages include functions that make decisions on placing applications or virtual machines onto specific physical machines. However, they do not dynamically adapt existing placements in response to a change, they do not dynamically scale resources for an application beyond a single physical machine, their support for global management objectives is limited, and, most importantly, they are limited in scalability (to some thousand nodes at most) due to their centralized underlying architecture. In this sense, the solutions in this paper outline a way to redesign the software layer that provides placement functions in these packages, so that the above mentioned objective for resource management can be achieved. Little public information is available regarding the underlying capabilities of the public cloud services provided by companies like Amazon [12], Google [13] and Microsoft [14]. We expect that the road maps for their platforms include design goals for application placement which are similar to ours.

This paper is based on our prior work on scalable resource management for cloud environments [7]. It uses the middleware architecture from that work, adapts the formalization of the resource allocation problem and reuses the concept of computing resource allocation policies through gossip protocols. The key contributions of this paper are as follows. First, we present a generic gossip protocol for resource management in cloud environments which can be instantiated for specific objectives. Second, we formalize the problem of minimizing power consumption through server consolidation and provide a heuristic solution in form of an instance of the generic protocol. Finally, we demonstrate through simulations the effectiveness of the protocol compared to an ideal system, and we show that the protocol scales well to a very large cloud.

The paper is structured as follows. Section II outlines the architecture of a middleware layer that performs resource management for a large-scale cloud environment. Section III presents our model for resource management in cloud environments and our generic solution to the problem of resource management. Section IV presents the specific problem studied in this paper and our proposed solution. The solution is evaluated through simulations in Section V. Section VI reviews related work, and Section VII contains the conclusion of this research and outlines of future work.

II. System Architecture

Figure 2 (left) shows the architecture of the cloud middleware. The components of the middleware layer run on all machines. The resources of the cloud are primarily consumed by module instances whereby the functionality of a site is made up of one or more modules. In the middleware, a module either contains part of the service logic of a site (denoted by m_i in figure 2) or a site manager (denoted by SM_i).

Each machine runs a *machine manager* component that computes the resource allocation policy, which includes deciding the module instances to run. The

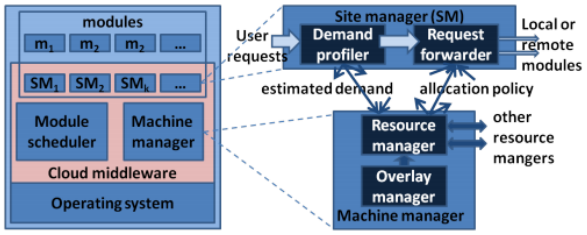


Fig. 2. The architecture for the cloud middleware (left) and components for request handling and resource allocation (right) [7].

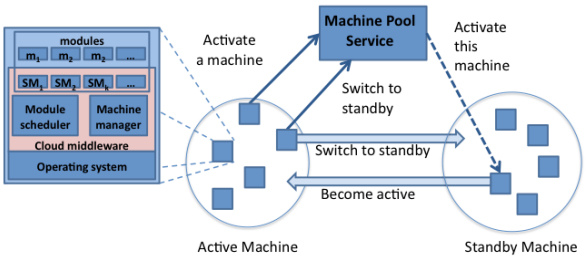


Fig. 3. Machine Pool Service

resource allocation policy is computed by a protocol (later in the paper called GRMP) that runs in the *resource manager* component. This component takes as input the projected demand for each module that the machine runs. The computed allocation policy is sent to the *module scheduler* for implementation/execution, as well as the *site managers* for making decisions on request forwarding. The *overlay manager* implements a distributed algorithm that maintains an overlay graph of the machines in the cloud and provides each resource manager with a list of machines to interact with.

Our architecture associates a site manager with each site. Each site manager handles user requests to a particular site. It has two important components: a *demand profiler* and *request forwarder*. The *demand profiler* estimates the resource demand of each module of the site based on the request statistics, QoS targets, etc. (Examples of such a profiler can be found in [15], [16].) The estimate is forwarded to all machine managers that run instances of modules belonging to the site. Similarly, the *request forwarder* sends user requests for processing to instances of modules belonging to the site. Request forwarding decisions take into account the resource allocation policy and constraints such as session affinity. Figure 2 (right) shows the components of a site manager and how they relate to machine managers.

From the point of view of power consumption, we consider a machine as having two states, *active* and *standby*. An active machine runs all software layers and components shown in figure 2 and, therefore, consumes a high level of power, while a standby machine does not execute any of the components in figure 2 and its power consumption is thus small or negligible. In this work we restrict ourselves to one standby state, for the reasons given in [17], knowing that the industry standard ACPI defines several levels of standby [18]. The standby state in our work can be realized as the ACPI G2 state in the ACPI specification. This is because the state allows an activation of a machine remotely through a wake-on-LAN packet. Each machine in the cloud is registered with the *machine pool service* shown in figure 3, which keeps track of the machine's power state, i.e., active or standby.

The resource manager component determines whether a machine can be put to standby or an additional machine needs to be activated. In the former case, it sends a *Switch-to-standby* message to the machine pool service, which subsequently switches the machine to the standby state. In the latter case, it sends an *Activate-a-machine* message to the service, which returns the identifier of an activated machine, if one is available.

The remainder of this paper focuses on the functionality of the resource manager component. For other components of our architecture, such as overlay manager and demand profiler, we rely on known solutions. A scalable design for the machine pool service is part of our future work.

III. Modeling Resource Allocation and our Generic Solution

For this work, we consider a cloud as having computational resources (i.e., CPU) and memory resources, which are available on the machines of the cloud infrastructure. We assume the machines in the cloud to be homogenous in the sense that their CPU and memory capacities as well as their power consumption properties are identical.

We restrict the discussion to the case where all machines belong to a single cluster, and cooperate as peers in the task of resource allocation. The specific problem we address is that of placing modules (more precisely: identical instances of modules) on machines and allocating cloud resources to these modules, such that the objectives of the cloud are achieved.

We model the problem of resource management as that of an optimization problem whose solution is a *configuration matrix* that controls the module scheduler and request forwarder components. At discrete points in time, events occur, such as load changes, addition and removal of site or machines, etc. In response to such an event, the optimization problem is solved again, in

order to keep the configuration optimal. We introduce our model for resource allocation in Section III-A and present the generic algorithm for resource management in Section III-B

A. The Model

We model the cloud as a system with a set of sites S and a set of machines N that run the sites. Each site $s \in S$ is composed of a set of modules denoted by M_s, and the set of all modules in the cloud is $M = \bigcup_{s \in S} M_s$.

We model the CPU demand as the vector $\boldsymbol{\omega}(t) = [\omega_1(t), \omega_2(t), \ldots, \omega_{|M|}(t)]^T$ and the memory demand as the vector $\boldsymbol{\gamma} = [\gamma_1, \gamma_2, \ldots, \gamma_{|M|}]^T$, assuming that CPU demand is time dependent while memory demand is not [19].

We consider a system that may run more than one instance of a module m, each on a different machine, in which case its CPU demand is divided among its instances. The demand $\omega_{n,m}(t)$ of an instance of m running on machine n is given by $\omega_{n,m}(t) = \alpha_{n,m}(t)\omega_m(t)$ where $\sum_{n \in N} \alpha_{n,m}(t) = 1$ and $\alpha_{n,m}(t) \geq 0$. We call the matrix A with elements $\alpha_{n,m}(t)$ the *configuration (matrix)* of the system. A is a non-negative matrix with $\mathbf{1}^T A = \mathbf{1}^T$.

A machine $n \in N$ in the cloud has a CPU capacity Ω and memory capacity Γ. We use $\boldsymbol{\Omega}$ and $\boldsymbol{\Gamma}$ to denote the vectors of CPU and memory capacities of all the machines in the system. An instance of module m running on machine n demands $\omega_{n,m}(t)$ CPU resource and γ_m memory resource from n. Machine n allocates to module m the CPU capacity $\hat{\omega}_{n,m}(t)$ (which may be different from $\omega_{n,m}(t)$) and the memory capacity γ_m. The value for $\hat{\omega}_{n,m}(t)$ depends on the allocation policy $\hat{\Omega}(t)$ in the cloud. The specific policy we use in this work allocates $\hat{\omega}_{n,m}(t) = \frac{\omega_{n,m}(t)}{\sum_i \omega_{n,i}} \Omega$.

Table I summarizes the notations used in this paper.

Notations

S, N, M	set of all sites, machines and modules
M_s, $s \in S$	set of modules of site s
$\boldsymbol{\omega}(t), \boldsymbol{\gamma} \in \mathbf{R}^{\lvert M \rvert}$	CPU and memory demand vectors
$A(t) \in \mathbf{R}^{\lvert N \rvert \times \lvert M \rvert}$	configuration matrix with elements $\alpha_{n,m}(t)$
$\hat{\Omega}(t) \in \mathbf{R}^{\lvert N \rvert \times \lvert M \rvert}$	CPU allocation matrix with elements $\hat{\omega}_{n,m}(t)$
$\Omega, \Gamma \in \mathbf{R}$	CPU and memory capacities of a machine
$P_n(t)$	power consumption of machine n

Formulas

$\sum_n \alpha_{n,m} = 1$	property of $\alpha_{n,m}$
$\omega_{n,m}(t) = \alpha_{n,m}(t)\omega_n$	resource demand of an instance of module m on machine n
$\hat{\omega}_{n,m}(t) = \frac{\omega_{n,m}(t)}{\sum_i \omega_{n,i}} \Omega$	the local resource allocation policy
$u_{n,m}(t) = \frac{\hat{\omega}_{n,m}(t)}{\omega_{n,m}(t)}$	utility of an instance of a module
$u(s,t) = \min_{n,m \in M_s} u_{n,m}(t)$	utility of a site s
$v_n(t) = \frac{\sum_m \omega_{n,m}(t)}{\Omega}$	relative CPU demand of a machine n
$g_n(t) = \frac{\sum_m \gamma_m(t)}{\Gamma}$	relative memory demand of a machine n
$CLF = \frac{\sum_m \omega_m(t)}{\lvert N \rvert \Omega}$	the CPU load factor of the cloud
$MLF = \frac{\sum_m \gamma_m(t)}{\lvert N \rvert \Gamma}$	the memory load factor of the cloud
$P^c(t) = \sum_n P_n(t)$	power consumption of the cloud

Evaluation metrics

$\frac{\lvert N \rvert - P^c}{\lvert N \rvert}$	reduction of power consumption
$\frac{\sigma\{u(s,t)\}_{s \in S}}{\mu\{u(s,t)\}_{s \in S}}$	fairness of resource allocation
$\frac{\lvert \{s\}_{u(s,t) \geq 1} \rvert}{\lvert S \rvert}$	satisfied demand

TABLE I
SUMMARY OF NOTATIONS, FORMULAS AND METRICS USED IN FORMALIZING AND EVALUATING RESOURCE ALLOCATION

B. GRMP: The Generic Resource Management Protocol

According to the above model, the configuration matrix A determines how cloud resources are allocated to sites. We advocate the use of a gossip protocol to efficiently compute this matrix for a large-scale cloud. Gossip protocols are round-based protocols where, in each round, a node selects a subset of other nodes to interact with. Node selection is often probabilistic and as nodes execute more rounds, their states converge to a desired state. Gossip protocols have been proposed for a number of management tasks including disseminating information in a robust way, computing aggregates, as well as creating and maintaining overlays.

In this subsection, we introduce a generic gossip protocol for resource allocation, which can be instantiated for various management objectives. We call this protocol GRMP (Generic Resource Management Protocol). GRMP runs in the Resource Manager component of all machines in the cloud (See Figure 2). The set of candidate machines to interact with is maintained by the Overlay Manager component of the Machine Manager.

GRMP is invoked at discrete points in time. Depending on the specific deployment, the invocation may be periodic, in response to a significant load change, or a combination of both. During each invocation of GRMP, each machine executes r_{max} rounds and outputs the configuration matrix A. The value for r_{max} depends on the specific instantiation of GRMP and is typically chosen such that the computed matrix A would not change 'significantly' as a result of further execution of rounds beyond r_{max}. The matrix A is distributed across the machines of the system and controls the start and stop of module instances and determines the control policies for module schedulers and request forwarders.

Algorithm 1 Protocol GRMP computes a configuration matrix A. Code for machine n

```
initialization
 1: read ω, γ, Ω, Γ, row_n(A);
 2: initInstance();
 3: start passive and active threads;
active thread
 1: for r = 1 to r_max do
 2:    n' = choosePeer();
 3:    send(n', row_n(A)); row_n'(A) = receive(n');
 4:    updatePlacement(n', row_n'(A));
 5:    sleep until end of round;
 6: write row_n(A);
passive thread
 1: while true do
 2:    row_n'(A) = receive(n'); send(n', row_n(A));
 3:    updatePlacement(n', row_n'(A));
```

The resource manager component determines whether the computed configuration matrix is implemented or not. We assume that the time it takes for GRMP to compute a new configuration, A, is small compared to the time between events that trigger consecutive runs of the protocols. At the time of initialization, GRMP reads as input a feasible configuration of the system, which can be computed using, e.g., [7], [20]. At later invocations, the protocol reads as input the configuration matrix produced during the previous run.

The pseudocode of GRMP is given in Algorithm 1. The protocol follows the so-called push-pull gossip interaction pattern, which we implement with an active and a passive thread on each machine. To keep the presentation simple, we omit thread synchronization primitives which prevent concurrent update of the local state by the active and passive threads.

GRMP is a generic protocol in the sense that three abstract methods must be implemented in order to compute a configuration matrix for a specific resource management objective.

1) $initInstance()$ is the initialization method for the specific gossip protocol.
2) $choosePeer()$ is the method for selecting a peer for gossip interaction.
3) $updatePlacement()$ is the method for recomputing the local state during a gossip interaction.

In subsection IV-B, we present an instantiation of GRMP, called GRMP-Q, which performs resource allocation for the objective of reducing power consumption. A gossip protocol that we developed in our earlier work can also be interpreted as an instantiation of GRMP [7]. That protocol implements the objective of fair allocation of CPU resources to sites. While the methods $initInstance()$ and $choosePeer()$ are implemented in a similar way as those for GRMP-Q, the semantics of $updatePlacement()$ is different. It updates the local states of interacting machines in such a way that their relative CPU demands, computed as $\frac{\sum_m \omega_{n,m}}{\Omega}$ for machine n, are equalized. This protocol is optimal under certain conditions, which means that the sequence of configurations the protocol generates when executing rounds converges exponentially fast to an optimal one [7].

IV. The Problem and our Solution

A. Resource Management as an Optimization Problem

The first objective is to satisfy the user demand if this is possible with the available cluster resources (i.e., underload) and to fairly allocate resource if it is not (i.e., overload). We formalize this using the concept of utility. We define the utility generated by an instance of module m on machine n as the ratio of the allocated CPU capacity to the demand of the instance on that particular machine, namely, $u_{n,m}(t) = \frac{\hat{\omega}_{n,m}(t)}{\omega_{n,m}(t)}$. (An instance with $\omega_{n,m} = 0$ generates a utility of ∞.) The utility generated by a site is defined as $u(s,t) = \min_{n,m \in M_s} u_{n,m}(t)$. The cloud utility $U^c(t)$ is then defined as $U^c(t) = \min_{s|u(s,t)\leq 1} u(s,t) = \min_{n,m|u_{n,m}\leq 1} u_{n,m}(t)$. The first objective can then be expressed as maximizing $U^c(t)$, which ensures that all site demands are satisfied in case of underload. In case of overload, maximizing $U^c(t)$ ensures *max-min fairness* regarding CPU resource allocation to sites.

The second objective is to minimize the power consumption of the cloud. We model the power consumption of a machine n with the function

$$P_n(t) = \begin{cases} 0 & \text{if } row_n(A)(t)\mathbf{1} = 0 \\ 1 & \text{otherwise} \end{cases}$$

$P_n(t) = 0$ means that the machine can be switched to standby state, and $P_n(t) = 1$ means that the machine must remain active. We express the power consumption of the cloud by $P^c(t) = \sum_n P_n(t)$. The second objective is therefore to minimize $P^c(t)$.

The problem of resource allocation is that of adapting a configuration $A(t)$ to a new configuration $A(t+1)$, such that the objectives of the resource management system are achieved for the new demand $\boldsymbol{\omega}(t+1)$. The third objective is to identify a configuration that minimizes a given cost function $c^*(A(t), A(t+1))$. This cost function captures the penalty associated with changing the configuration $A(t)$ to $A(t+1)$. Such a penalty may reflect, for example, a high level of network bandwidth consumption or a long service interruption time during

reconfiguration. (The cost function we consider in this work counts the number of module instances that are started to reconfigure the system from the current to the new configuration.)

We now formalize the optimization problem using the three objectives discussed above. Consider a cloud with CPU capacity $\mathbf{\Omega}$ and memory capacity $\mathbf{\Gamma}$. Then, given a configuration $A(t)$, CPU demand vector $\boldsymbol{\omega}(t+1)$ and memory demand vector $\boldsymbol{\gamma}$, the problem is to find a configuration $A(t+1)$ that solves the following optimization problem.

$$\begin{array}{ll} \text{maximize} & U^c(t+1) \\ \text{minimize} & P^c(t+1) \\ \text{minimize} & c^*(A(t), A(t+1)) \\ \text{subject to} & A(t+1) \geq 0,\ \mathbf{1}^T A(t+1) = \mathbf{1}^T \\ & \hat{\Omega}(A(t+1), \boldsymbol{\omega}(t+1))\mathbf{1} \preceq \mathbf{\Omega} \\ & \mathbf{sign}(A(t+1))\boldsymbol{\gamma} \preceq \mathbf{\Gamma}. \end{array} \quad \text{(OP)}$$

This optimization problem has prioritized objectives. This means that, among all configurations A that maximize the cloud utility U^c, we select those configurations that minimize the power consumption P^c. Out of these configurations, we choose one that minimizes the cost function c^*. The constraints of (OP) relate to (1) splitting up the CPU demand of each module into the demand of the module instances, and (2) ensuring that the allocated CPU and memory resources on each machine can not be larger than its available capacity.

Let us briefly comment on the hardness of (OP). Memory demand for a module is not divisible, which means that the memory demand of a module can not be split among its instances that run on different machines. This makes (OP) NP-hard. However, in many practical cases where the combined memory demand is significantly smaller than the memory capacity of the cloud, a solution to (OP) can easily be found.

B. Our Solution GRMP-Q: A Heuristic Solution to (OP)

As an instance of GRMP, GRMP-Q implements the three abstract methods of GRMP as shown in algorithm 2. In the $initInstance$ method, the machine n initializes N_n, the set of machines that run common modules with n. A machine n prefers to run the gossip step with an other machine $j \in N_n$. The reason is that load can be moved between the two machines without requiring additional memory and at no cost of reconfiguration. However, always selecting j from N_n may result in the cloud being partitioned into disjoint sets of interacting machines. To avoid this situation, n is occasionally paired with a machine outside of the set N_n. The neighbor selection function $choosePeer()$ implements this as follows: it returns a machine selected uniformly at random from the set N_n with some (configurable) probability p and from the set $N - N_n$ with probability $1 - p$.

The core of the protocol is implemented in the $updatePlacement$ function that moves module instances from one machine to another. The objective of the movement is determined by the *relative CPU demand* of the participating machines, which is defined for machine n as $v_n = \sum_m \omega_{n,m}/\Omega$. Specifically, for machines n and j, if $v_n + v_j \geq 2$, the protocol estimates that the cloud is in overload and calls a function that aims to achieve fairness for CPU resources. This function (that is outlined in [7]) moves modules from the machine with higher relative demand to the machine with lower relative demand, with the goal of equalizing v_n and v_j.

If $v_n + v_j < 2$, the protocol estimates that the cloud is in underload and calls functions that aim to reduce the power consumption of the cloud, while ensuring demands of sites are satisfied. These functions are $packNonShared$, which is always called, and $packShared$ which is called only if the two machines share modules. The functions are based on the following two concepts.

The first concept, which is implemented by the function $pickSrcDest$, ensures that the protocol primarily moves modules from an overloaded machine to the underloaded one, aiming to satisfy the demand of the module instances on the overloaded machine. On the other hand, if both machines are underloaded, the protocol moves modules from the machine with lower load to the machine with higher load, in an attempt to fully pack one machine or freeing up another.

The second concept relates to the packing efficiency of the protocol. Specifically, it attempts to avoid situations where a single type of resource (i.e., only CPU or memory) of a machine is utilized while the other is not. Such a situation reduces the packing efficiency of the protocol and hence the reduction in power consumption. Therefore, during an interaction, the protocol identifies the dominant resource at the destination (i.e., the resource type that has the larger relative demand), and chooses modules at the source machine such that they have less of the dominant resource. (In the pseudocode, the *relative memory demand* is defined as $g_n = \sum_m \gamma_m/\Gamma$.)

The function $pickSrcDest$ returns a pair $(src, dest)$ where $dest$ is the machine should be packed with modules migrated from src. The function takes input the gossiping machine j and makes a decision based

on the relative CPU demand of n and j. The function first picks $dest$ and src to be the machines with higher and lower relative CPU demands respectively. Then, if $dest$ is overloaded, it swaps $dest$ and src. Note that it is impossible for both n and j to be overloaded with CPU demand in this function because of the preliminary check in the $updatePlacement$ method discussed earlier.

The $packShared$ method focuses on packing shared modules. The memory constraint in the destination machine is not a concern here because moving the demand of shared modules does not increase memory consumption. However, the movement increases CPU consumption in the destination machine. Hence, we try to minimize CPU consumption in the destination while maximizing the amount of memory freed in the source machine, which we achieve by sorting the modules with decreasing $\gamma_{s,m}/\omega_{s,m}$. ($\alpha_{n,m} = 0$ implies stopping module m on machine n.)

CPU demand to be migrated from the source machine, $\Delta\omega_s$, depends on whether the source machine is overloaded with CPU demand ($v_s > 1$). If the source machine is overloaded, the amount is the excess demand with respect to the capacity. On the other hand, if the source machine is not overloaded, the amount is the total CPU demand. This separation of $\Delta\omega_s$ ensures that the amount of CPU demand to migrate is heuristically low whether or not the source is overloaded. The protocol progresses until either (1) it runs out of shared modules or (2) the amount of CPU demand to migrate from the source machine is met or (3) the free CPU capacity on the destination machine is exhausted.

The $packNonShared$ method considers the case where gossiping machines do not share modules. The CPU demand to be migrated from the source machine is computed in the same way as the $packShared$ method explained above. However, in contrast to the $packShared$ method, moving modules increases the memory consumption at the destination. In order to fill up both CPU and memory capacities on the destination machine, modules to be moved are selected from the list of modules on the source, sorted with the sorting criteria $sortCri$. $sortCri$ depends on the workload conditions of the destination machine. If the workload is CPU-intensive (i.e., $v_d \geq g_d$), the modules to be migrated should be memory-intensive or vice versa. The method iterates until either (1) there are no more modules on the source or (2) the amount of CPU demand to migrate from the source machine is met or (3) the free memory/CPU capacity of the destination machine is used up.

Algorithm 2 Protocol GRMP-Q, an instance of GRMP for solving (OP). Code for machine n

```
initInstance()
 1: read N_n;
choosePeer()
 1: if rand(0..1) < p then
 2:   return unifrand(N_n);
 3: else
 4:   return unifrand(N − N_n);
updatePlacement(j, row_j(A))
 1: if (v_n + v_j ≥ 2) then
 2:   equalize(j, row_j(A));
 3: else
 4:   if j ∈ N_n then
 5:     packShared(j);
 6:   packNonShared(j);
packShared(j)
 1: (s, d) = pickSrcDest(j); Δω_d = Ω − Σ_m ω_{d,m};
 2: if v_s > 1 then Δω_s = Σ_m ω_{s,m} − Ω; else
    Δω_s = Σ_m ω_{s,m};
 3: Let mod be the list of modules
    shared by s and d, sorted by
    decreasing γ_{s,m}/ω_{s,m};
 4: while mod ≠ ∅ ∧ Δω_s > 0 ∧ Δω_d > 0 do
 5:   m = remove first element from mod;
 6:   δω = min(Δω_d, Δω_s, ω_{s,m}); Δω_d−=δω;
 7:   Δω_s−=δω; δα=α_{s,m} δω/ω_{s,m}; α_{d,m}+=δα;
      α_{s,m}−=δα;
packNonShared(j)
 1: (s, d) = pickSrcDest(j);
 2: Δγ_d = Γ − Σ_m γ_{d,m}; Δω_d = Ω − Σ_m ω_{d,m};
 3: if v_s > 1 then Δω_s = Σ_m ω_{s,m} − Ω; else
    Δω_s = Σ_m ω_{s,m};
 4: if v_d ≥ g_d then sortCri = γ_{s,m}/ω_{s,m}; else
    sortCri = ω_{s,m}/γ_{s,m};
 5: Let mod be the list of modules
    on s not shared with d, sorted by
    decreasing sortCri;
 6: while mod ≠ ∅ ∧ Δγ_d > 0 ∧ Δω_d > 0 ∧ Δω_s > 0
    do
 7:   m = remove first element from mod;
 8:   δω = min(Δω_s, Δω_d, ω_{s,m}); δγ = γ_{s,m};
 9:   if Δγ_d ≥ δγ then
10:     δα = α_{s,m} δω/ω_{s,m}; α_{d,m}+=δα; α_{s,m}−=δα;
11:     Δγ_d −= δγ; Δω_d −= δω; Δω_s −= δω;
pickSrcDest(j)
 1: dest=arg max(v_n, v_j); src=arg min(v_n, v_j);
 2: if v_dest > 1 then swap dest and src;
 3: return (src, dest);
```

C. Properties of GRMP-Q

Since GRMP-Q is a heuristic solution, the configuration it produces is generally not optimal in the sense of (OP). To understand the properties of the protocol, we introduce useful notions: CPU load factor $CLF = \frac{\boldsymbol{\omega}^T \mathbf{1}}{|N|\Omega}$

and memory load factor $MLF = \frac{\boldsymbol{\gamma}^T \mathbf{1}}{|N|\Gamma}$. The cloud is in overload whenever $CLF > 1$, which means that the total demand for CPU resources exceeds the available capacity in the cloud. (This paper does not consider the case $MLF > 1$ because an initial placement for such a load in the cloud is not possible and memory demands are assumed to be constant.)

a) Cloud in overload ($CLF > 1, MLF < 1$): The protocol is designed in a way that all machines in the cloud eventually become overloaded. Once this is the case, the protocol executes in the same way as the fairness protocol described in [7], which means that it attempts to allocate CPU resource across sites using a max-min fairness policy.

b) Memory demand much smaller than capacity ($MLF \ll 1$): After each gossip interaction, the interacting machines are in one of the following states: (1) both machines have equal load. (2) one machine carries maximum CPU load. (3) one machine carries no load. Under these conditions, the configuration computed by the protocol converges to an optimal solution of (OP) — if we neglect the cost of reconfiguration. If $CLF < 1$, an optimal solution implies that $\lfloor |N|CLF \rfloor$ machines carry maximum load, $|N| - \lceil |N|CLF \rceil$ carry no load, while all site demands are satisfied.

c) General case ($CLF < 1, MLF < 1$): By design, the protocol gives preference to moving load away from an overloaded machine over transferring load for the purpose of reducing power consumption. As a consequence, we can state that, if the new configuration the protocol produces includes machines that do not carry load, the machines with load fully satisfy the demand.

V. Evaluation through Simulation

We have evaluated GRMP-Q through extensive simulations using a discrete event simulator that we developed in-house. We simulate a distributed system that runs the machine manager components of all machines in the cloud. Specifically, these machine managers execute the protocol GRMP-Q, which computes the allocation matrix A, and also the CYCLON protocol, which provides for GRMP-Q the function of selecting a random neighbor. The simulator also implements the algorithm outlined in [7] to compute an initial feasible configuration of the cloud. The external events for this simulation are the changes in the demand vector $\boldsymbol{\omega}$.

Evaluation metrics: We measure the *reduction of power consumption* as $\frac{|N|-P^c}{|N|}$, the fraction of machines in the cloud that are freed by the protocol. Second, we measure the *fairness* of resource allocation through the coefficient of variation of site utilities, computed as the ratio of the standard deviation to the average of the utilities. Third, we measure the *satisfied demand* as the fraction of sites that generate utilities of larger or equal to 1. Finally, we measure the *cost of reconfiguration* as the ratio of module instances started to module instances running, per machine.

Generating the demand vectors $\boldsymbol{\omega}$ and $\boldsymbol{\gamma}$: The number of modules of a site is chosen from a discrete Poisson distribution with mean 1, incremented by 1. The memory demand of a module is chosen uniformly at random from the set $c_\gamma \cdot \{$128MB, 256MB, 512MB, 1GB, 2GB$\}$. For a site s, at each change in demand, the demand profiler generates CPU demands chosen from an exponential distribution with mean $\omega(s)$. We choose the distribution for $\omega(s)$ among all sites to be Zipf distributed with $\alpha = 0.7$, following evidence in [22]. The maximum value for the distribution is $c_\omega \cdot$500G CPU units and the population size used is 20,000. For a module m of site s, we choose a demand factor β_m with $\sum_{m \in M_s} \beta_m = 1$, chosen uniformly at random, which describes the share of module m in the demand of the site s. c_γ and c_ω are scaling factors (see below).

Scenario parameters: We evaluate the performance of our resource allocation protocol GRMP-Q under varying intensities of CPU and memory load, which we vary by changing c_γ and c_ω. All machines in the cloud have the same CPU capacity 34.513G CPU units and memory capacity 36.409 GB. These values give $MLF = CLF = 0.5$ for $c_\gamma = c_\omega = 5$. We use the following parameters unless stated otherwise:

- $|N|$=10,000, $|S|$=24,000, $r_{max} = 30$, $p = \frac{|N_n|}{1+|N_n|}$
- maximum number of instances/module: 100, number of load changes during a run: 100

A. Performance of GRMP-Q under Varying CLF and MLF

In this scenario, we evaluate the performance of GRMP-Q for CLF=$\{$0.1,0.4,0.7,1.0,1.3$\}$ and MLF=$\{$0.1,0.3,0.5,0.7,0.9$\}$, by measuring the metrics listed above. We compare our results with that of an ideal system that has the aggregate CPU and memory capacity of the cloud and that consumes power according to the function $P^C_{lb} = \lceil \min(1, \max(CLF, MLF)) \rceil$. ($P^C_{lb}$ is a lower bound to P^C which is a good approximation of the optimal value for P^C, for low values of MLF.)

1) Reduction of Power Consumption: Figure 4a presents the potential reduction of power consumption by GRMP-Q for various values of CLF and MLF. As expected, this quantity decreases for increasing CLF and MLF. For instance, it decreases from 85% for $CLF = MLF = 0.1$ to 0 for $CLF \geq 1$ and $MLF \geq 0.9$. This is expected since the number of

machines needed to run and satisfy the demands of all sites increases with both CLF and MLF. The potential reduction in power consumption reduces to 0 for $CLF > 1$ as the demand can not be satisfied, even with all the machines in the cloud.

2) Satisfied demand: Figure 4b suggests that satisfied demand depends on both CLF and MLF. For the ideal system, the satisfied demand depends only on CLF. Specifically, the demand of all sites is satisfied when CLF is less than 1 and not satisfied otherwise. Our protocol satisfies more than 99% of site demands in underload scenarios, except for the case of $(CLF, MLF) = (0.7, 0.7)$ and $(CLF, MLF) = (0.7, 0.9)$. As can be seen, for CLF values larger than 1, our protocol achieves a larger satisfied demand than the ideal system, at the expense of an unfair CPU allocation.

3) Fairness: Figure 4c presents the measured fairness property of our protocol. First, we observe that for the underload conditions, the fairness metric is very high, indicating that the resource allocation is unfair. This happens because when the protocol tries to minimize number of active machines, there are some machines which are not completely packed with modules. Modules on such machines would be allocated much more resources compared to the modules in the packed machines and thus have higher utilities. Note that even though the resource allocation is unfair, the protocol would achieve its goal as long as all site demands are satisfied.

Second, we observe that the fairness metric is much lower in overload conditions. This suggests that, in line with the goal of our protocol, resource allocation is fair among sites. In addition, one can observe that the fairness metric increases with MLF. In particular, fairness increases from 0.02 when MLF is 0.1 to 0.58 when MLF is 0.9. This is also to be expected since achieving fairness becomes hard when MLF is high.

4) Cost of reconfiguration: Figure 4d indicates that the cost of reconfiguration is generally independent of MLF but grows with CLF. This is attributed to the increase in the number of active machines in the cloud as CLF increases. There is an exception when MLF is extremely high ($MLF = 0.9$) where the cost drops. This occurs because at $MLF = 0.9$, there is not much memory to allow moving modules around. Beyond $CLF = 0.7$, almost all machines become active (see figure 4a), which results in the cost remaining constant.

B. Scalability

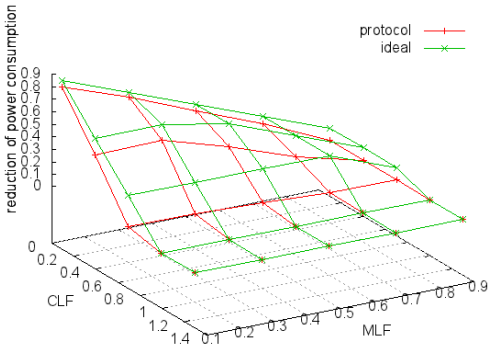
In this scenario, we measure the dependence of our evaluation metrics on the size of the cloud. To achieve this, we run simulations for a cloud with (2,500, 5,000, 10,000, 20,000, 40,000, 160,000) machines and (6,000, 12,000, 24,000, 48,000, 96,000, 384,000) sites respectively (keeping the ratio of sites to machines at 2.4). In the setting, we evaluate two different sets of CLF and MLF which are $\{(0.5, 0.5), (0.25, 0.25)\}$. Figure 5 shows the result obtained, which indicates that all metrics considered are independent of the system size. In other words, if the number of machines grows at the same rate as the number of sites, (while the CPU and memory capacities of a machine, as well as all parameters characterizing a site, such as demand, number of modules, etc., stay the same), we expect all considered metrics to remain constant. Note that our conclusion is related exclusively to the scalability of the protocol GRMP-Q. The complete resource management system includes many more functions that have not been evaluated here, for instance, the scalability of effectively choosing a random peer.

VI. Related work

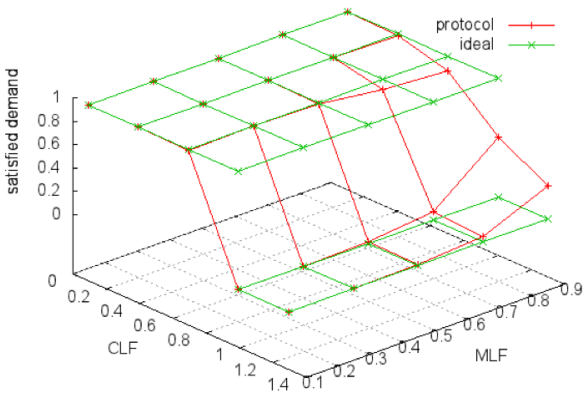
The problem of reducing power consumption of a datacenter under performance constraints has been extensively studied [5], [23]–[27] and there are also products that incorporate such features [4]. The key differentiating factor of our work, in relation to all other works presented in the literature is that we present a highly scalable resource management system that uses decentralized algorithms to implement the key functionalities of the resource management process. In contrast, the works outlined below rely on centralized controllers that are responsible for computing the resource allocation policy of the whole system. This centralized approach, coupled with the combinatorial nature of the server consolidation problem does not scale beyond thousands of machines.

The work that is probably closest to us is the one presented in [24]. There, the authors present Mistral, a framework for dynamically managing power, performance and adaptation costs. They model the problem of resource allocation as that of a utility maximization problem and propose a search based heuristic which they evaluate on a testbed. The authors claim that their approach is scalable, since it can be used in a hierarchical manner. However, they do not provide any evidence to support this and the reported measurement results indicate otherwise.

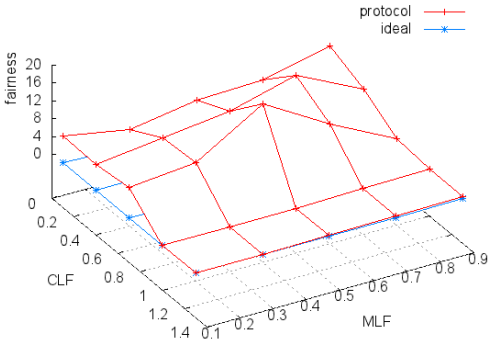
The works in [23], [26] look at the problem of reducing power consumption of a datacenter by considering both power consumed by IT equipment and the power dissipated by the cooling system of the datacenter. Our work only focuses on reducing the server power. The extension that relates to cooling power is a functionality that shall be considered in the design of the machine pool service in our future work.



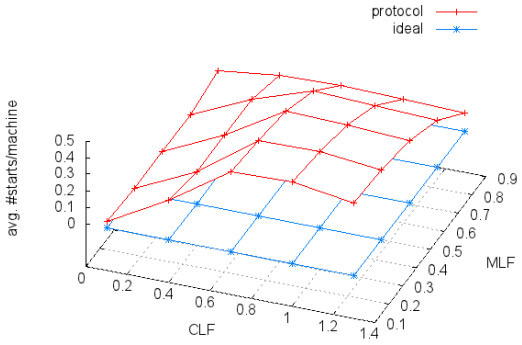
(a) Fraction of machines that can be put to standby



(b) Fraction of sites with satisfied demand.



(c) Fairness among sites (0 means optimal fairness)



(d) Cost of change in conguration over all machines.

Fig. 4. The performance of the resource allocation protocol GRMP-Q in function of the CPU load factor (CLF) and the memory load factor (MLF) of the cloud (10,000 machines, 24,000 sites).

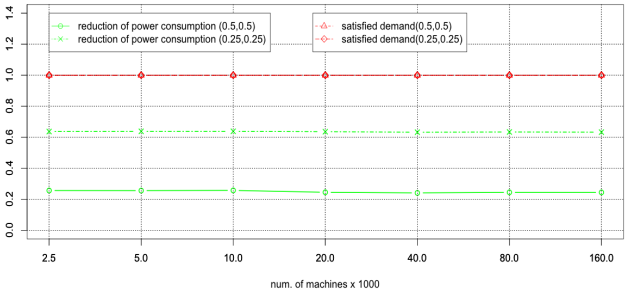


Fig. 5. Scalability with respect to the number of machines and sites.

[27] discusses a trace-based solution for the dynamic server consolidation problem. Similar to our work, the solution considers CPU and memory resources and takes into account migration overhead. The performance of their solution is evaluated with real trace that is obtained from 138 SAP enterprise applications. In contrast, we use synthetic workload to evaluate the performance of our system. To the best of our knowledge, there is no publicly available workload trace in a cloud environment that is suitable to the sizes of system we consider in this work.

The works in [5], [28] present static server consolidation approaches that attempt to exploit the specific scenarios they are deployed in. In [5], the authors analyze the workload of the applications in a given datacenter in order to compute the best static consolidation strategy that minimizes the power consumption without adversely affecting the performance of the running applications. In [28], the authors present a consolidation strategy that takes advantage of the min-max and shares-based CPU

resource allocation approach in Xen and VMware.

The works in [25], [29] present a formal treatment for some variants of the static server consolidation problem and present heuristic solutions that have provable properties with respect to how close they are to an optimal one. The experimental evaluation of these solutions shows that, it is difficult to solve the consolidation problem for more than a thousand of machines.

VII. Discussion and conclusion

We make three contributions with this paper. First, we introduce and formalize the problem of minimizing power consumption through server consolidation when the system is in underload and fair resource allocation in case of overload. Second, we present GRMP, a generic gossip protocol for resource management that can be instantiated for different objectives. (A protocol for fair resource allocation from our earlier work is in fact an instantiation of this protocol.) Finally, we present an instance of GRMP that provides a heuristic solution to the problem of minimizing power consumption, which we show to be effective and scalable.

The simulation studies of GRMP-Q indicate that the protocol performs in accordance with its design goals stated in section I, for the parameter ranges investigated. For instance, in an underload scenario with $CLF = MLF = 0.1$, the protocol computed a configuration where less than 20% of the machines carry load, while still satisfying user demand. In overload scenarios, the protocol allocates resources fairly to sites, as long as sufficient memory is available. Furthermore, the results demonstrate that the protocol is scalable in the sense that its key performance metrics do not change with increasing system size.

With respect to future work, we plan to (1) determine the convergence rate of GRMP-Q and its dependence on CPU and memory demands; (2) develop a version of the protocol for a heterogeneous cloud environment in which CPU and memory capacities vary across machines; (3) develop a distributed mechanism that efficiently places new sites; (4) make the protocol robust to machine failures; (5) develop versions of GRMP that support further objectives (e.g., service differentiation) and constraints (e.g., colocation and anti-colocation); (6) develop a scalable implementation of the machine pool service that considers power consumed for cooling.

References

[1] U.S. EPA, "Report to congress on server and data center energy efficiency public law 109-431," 2007.

[2] The Climate Group, "Smart 2020: Enabling thelow carbon economy in theinformation age," June 2008.

[3] Open Data Center Alliance, "Open data center alliance usage:carbon footprint values," June 2011.

[4] VMWare, "Vmware® distributed power management white paper," http://www.vmware.com/files/pdf/DPM.pdf.

[5] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari, "Server workload analysis for power minimization using consolidation," in *USENIX'09*. Berkeley, CA, USA: USENIX Association, 2009, pp. 28–28.

[6] U.S. EPA, "Working group notes from the EPA technical workshop on energy efficient servers and datacenters," 2007.

[7] F. Wuhib, R. Stadler, and M. Spreitzer, "Gossip-based resource management for cloud environments," in *CNSM 2010*, October, pp. 1 –8.

[8] OpenNebula Project Leads, "http://opennebula.org/."

[9] Eucalyptus Systems, Inc., "http://www.eucalyptus.com/."

[10] UC Santa Barbara, "http://appscale.cs.ucsb.edu/."

[11] VMWare, "http://www.cloudfoundry.com/."

[12] Amazon Web Services LLC, "http://aws.amazon.com/ec2/."

[13] Google Inc., "http://code.google.com/appengine/."

[14] Microsoft Inc., "http://www.microsoft.com/windowsazure/."

[15] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Dynamic estimation of CPU demand of web traffic," in *ValueTools 2006*. New York, NY, USA: ACM, p. 26.

[16] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in *CNSM 2010*, October, pp. 9 –16.

[17] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: eliminating server idle power," *SIGPLAN Not.*, vol. 44, pp. 205–216, March 2009.

[18] Hewlett-Packard, Intel, Microsoft, Phoenix Technologies Ltd., Toshiba Corporations, "Advanced configuration and power interface specification," 2010.

[19] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade, "Utility-based placement of dynamic web applications with fairness goals," in *IEEE NOMS*, April 2008, pp. 9 –16.

[20] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *WWW2007*. New York, NY, USA: ACM, 2007, pp. 331–340.

[21] F. Wuhib, R. Stadler, and M. Spreitzer, "Gossip-based resource management for cloud environments (long version)," KTH Royal Institute of Technology, Tech. Rep., August 2010, tRITA-EE 2010:032.

[22] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," in *INFOCOM*, vol. 1, 1999, pp. 126 –134.

[23] V. Petrucci, O. Loques, and D. Mossé, "Dynamic optimization of power and performance for virtualized server clusters," in *ACM SAC 2010*, 2010, pp. 263–264.

[24] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures," in *ICDCS2010*, 2010, pp. 62 –73.

[25] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE TSC*, vol. 3, no. 4, pp. 266 –278, 2010.

[26] N. Tolia, Z. Wang, P. Ranganathan, C. Bash, M. Marwah, and X. Zhu, "Unified thermal and power management in server enclosures," *ASME Conference Proceedings*, vol. 2009, no. 43604, pp. 721–730, 2009.

[27] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper, "An integrated approach to resource pool management: Policies, efficiency and quality metrics," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, june 2008, pp. 326 –335.

[28] M. Cardosa, M. Korupolu, and A. Singh, "Shares and utilities based power consolidation in virtualized server environments," in *IM 2009*, pp. 327 –334.

[29] C. Subramanian, A. Vasan, and A. Sivasubramaniam, "Reducing data center power with server consolidation: Approximation and evaluation," in *HiPC 2010*, 2010, pp. 1 –10.