Energy-Aware Autonomic Resource Allocation in Multitier Virtualized Environments

Danilo Ardagna, Barbara Panicucci, Marco Trubian, and Li Zhang

Abstract—With the increase of energy consumption associated with IT infrastructures, energy management is becoming a priority in the design and operation of complex service-based systems. At the same time, service providers need to comply with Service Level Agreement (SLA) contracts which determine the revenues and penalties on the basis of the achieved performance level. This paper focuses on the resource allocation problem in multitier virtualized systems with the goal of maximizing the SLAs revenue while minimizing energy costs. The main novelty of our approach is to address—in a unifying framework—service centers resource management by exploiting as actuation mechanisms allocation of virtual machines (VMs) to servers, load balancing, capacity allocation, server power state tuning, and dynamic voltage/frequency scaling. Resource management is modeled as an NP-hard mixed integer nonlinear programming problem, and solved by a local search procedure. To validate its effectiveness, the proposed model is compared to top-performing state-of-the-art techniques. The evaluation is based on simulation and on real experiments performed in a prototype environment. Synthetic as well as realistic workloads and a number of different scenarios of interest are considered. Results show that we are able to yield significant revenue gains for the provider when compared to alternative methods (up to 45 percent). Moreover, solutions are robust to service time and workload variations.

Index Terms—Performance attributes, performance of systems, quality concepts, resource allocation, optimization, energy cost reduction.

1 Introduction

Power consumption and energy efficiency are becoming very important in the design and management of computer systems. Due to the growth in the number of servers and the increasing complexity of the network infrastructure, energy costs are quickly rising in large-scale service centers. In [12], it is shown that service centers consume 1.5 percent of the power produced in the US, and will reach 4.5 percent within five years. Moreover, energy use in service centers is starting to prompt environmental concerns in terms of CO2 emissions. A sustainable and power aware computing system needs to provide services with a tradeoff between performance and energy consumption: resource management should not be solely focused on performance but also equally take energy efficiency into account. The aim of this paper is to develop energy-aware resource allocation policies for web systems. The goal is to provide services which achieve Quality of Service (QoS) requirements, while minimizing the energy consumption of the computing infrastructure. In this context, QoS requirements, formally stipulated in Service Level Agreements (SLAs) contracts, are difficult to satisfy due to the high variability of Internet workloads, which may vary by several order of magnitude within the same business day [8]. Planning the capacity for the worst case scenario is either infeasible or extremely inefficient. To handle workload variations, many service centers employ autonomic self-managing techniques: resources are dynamically allocated among running applications on the base of short-term demand estimates, e.g., [33], [5], [32]. The goal is to meet the application requirements while adaptively controlling the IT infrastructure. This leads to the problem of an efficient use of the resources and the reduction of energy consumption. Modern service centers host multitier applications in virtualized environments. Physical resources (e.g., CPU, disks, and communication network) are partitioned into multiple virtual ones, creating isolated virtual machines (VMs) each running at a fraction of the physical system capacity. Autonomic self-managing techniques are implemented by network controllers which can establish the set of applications executed by each server (i.e., application placement problem), the request volumes at various servers (i.e., load balancing problem), and the capacity devoted for the execution of each application at each server (i.e., capacity allocation problem). Network controllers can also decide to switch servers into low power sleep state depending on the system load (i.e., server switching problem) or to reduce the frequency of operation of servers (i.e., frequency scaling problem). Indeed, modern servers allow to dynamically change the supply voltage and CPU operating frequency exploiting Dynamic Voltage and Frequency Scaling (DVFS) mechanisms [19], [17].

These problems are taken with different time scales. Since load balancing, capacity allocation, and frequency scaling problems have small time requirements, they are performed every few minutes (short-term planning). Instead, server switching and application placement problems are taken about every half an hour because they introduce a
more significant system overhead (long-term planning). To the best of our knowledge, so far each of these problems has been addressed separately. However, it is noteworthy that the solutions of each problem are closely related. For example, the request volume determined for a given application at a given server could not be independent of the capacity allocated to that application on the server.

The main contribution of this paper is to integrate all the previous problems in a unifying framework, providing very efficient and robust solutions for the short-term and long-term problems. The relevance and novelty of our approach is that, unlike the previous literature, we consider the control variables jointly and this allows exploiting the interrelations among resource allocation decisions. This leads to a better tradeoff between performance and energy consumption with respect to the solutions obtained by facing the problems separately. The resource allocation problem is modeled as a mixed integer nonlinear programming problem and is solved using nonlinear programming techniques and a local search procedure. We demonstrate the effectiveness of our approach by simulation and performing tests on a real prototype environment. Extensive experiments show that, compared to state-of-the-art resource management approaches [22], [30], [32], our model can yield a significant revenue increase (up to 45 percent) for the provider by using lower resources and it is more robust to workload fluctuations.

The remainder of the paper is organized as follows: Section 2 reviews the approaches proposed in the literature. Section 3 introduces the system under study and the related performance model. The optimization problem formulation and the local search approach are presented in Sections 4 and 5. Section 6 is dedicated to the experimental results. Conclusions are finally drawn in Section 7.

2 RELATED WORK

The reduction of energy use in modern IT systems is receiving great attention by the research community. A lot of work has been done to achieve power reduction at the device level, e.g., in mobile systems to extend battery life. Nowadays, low power techniques and energy savings mechanisms are progressively introduced in server environments. All major industrial players are taking a position in the green IT arena (see, e.g., IBM’s project Big Green1 and HP’s Green up initiative2), while autonomic techniques have been developed to manage workload fluctuations and to determine optimal tradeoffs between performance and energy costs.

In the literature, five main problems have been considered to manage system allocation policies:

1. application placement,
2. admission control,
3. capacity allocation,
4. load balancing, and
5. energy consumption minimization.

Three main approaches have been proposed to address each of these problems: 1) the feedback loop based on control theory, 2) machine learning adaptive techniques, and 3) utility-based optimization techniques. In the following, we provide a description of these approaches. The main advantage of the feedback loop based on control theory is that it guarantees system stability. Furthermore, upon a change in workloads, these techniques can accurately model transient behavior and can adjust the system configuration within the time frame of a transitory, which can be fixed at design time. However, these techniques are typically implemented by local controllers and, hence, only local optimization objectives can be reached [1]. Most control theoretic approaches adopt system identification techniques to build linear time invariant models and then apply classical proportional integral differential control. In [23], a linear parameter varying framework has been proposed for the performance control of webservers by adopting DVFS as control variable. Authors in [19], [20] have implemented a limited lookahead controller which can determine the set of servers in active state, the corresponding operating frequency, and the allocation of VMs to physical servers. However, the implementation considers the VM placement and capacity allocation problems separately and the scalability of the approach has not been demonstrated. Recently, Kalman filters have been adopted to track and control the CPU utilization in virtualized environments in order to guide capacity allocation [16]. However, only a single multitier application executed in a single physical host has been considered. Finally, in [24], a control-oriented framework has been proposed that is able to coordinate different kinds of power managers ranging from the control of a single CPU to a cluster of blades.

Machine learning techniques base their learning sessions on live systems, without a need for an analytical model of the system. To maximize the revenue from SLAs, in [31], a capacity allocation technique that determines the assignment of applications to physical servers has been proposed. In [17], the authors have applied machine learning to coordinate multiple autonomic managers with different goals. In particular, their work has integrated an autonomous performance manager with a power manager in order to satisfy performance constraints while minimizing energy expenses exploiting server frequency scaling. A recognized advantage of machine learning techniques is that they accurately capture system behavior without any explicit performance or traffic model and with little built-in system-specific knowledge. However, training sessions tend to extend over several hours [17]. Furthermore, the proposed techniques usually consider a limited set of managed applications and apply actuation mechanisms separately.

Utility-based approaches have been introduced to optimize the degree of user satisfaction by expressing their goals in terms of user-level performance metrics. Typically, the system is modeled by means of a performance model embedded within an optimization framework. Optimization can provide global optimal solutions or suboptimal solutions by means of heuristics, depending on the complexity of the optimization model. Optimization is typically applied to each one of the five problems separately. Some research studies deal with admission control for overload protection of servers [33], [15]. Capacity allocation is typically viewed as a separate optimization activity that operates under the assumption that servers are protected from overload. Other studies focus on service

differentiation in server farms by physically partitioning the farm into separate clusters, each serving one of the request classes, e.g., [32]. New technology trends advocate the sharing of physical resources among multiple request classes [5], [29], [30] supported also by virtualization and server consolidation technologies. In [29], a virtualization framework able to support heterogeneous workloads (batch and web interactive system) is presented, but energy saving policies are not considered. Authors in [38] have presented a multilayer and multitime scale solution for the management of virtualized systems, but they do not consider the tradeoff between performance and system costs. The work in [35] faces with the problem of admission control and capacity allocation in multitier virtualized environment; however, the solutions are not integrated and only homogeneous systems with a single physical host per tier are considered. Finally, in [18], a framework able to coordinate independent resource managers running on distributed physical nodes of a virtualized data center has been presented. Note that, autonomic management today is implemented also in virtualization products, e.g., VMWare DRS or Virtuoso VSched [29], with the aim at equalizing the use of resources instead of determining performance and energy tradeoff. Considering the energy consumption optimization, authors in [25] have presented the result of a server consolidation project on blade servers based on a power budget mechanism. Processors’ frequencies are throttled in order to achieve CPU utilization and energy savings goals, but the approach cannot provide any SLA guarantee a priori. Similarly, the work in [11] provides power budget policies for virtualized environments proposing also an accurate model to predict server system average power consumption.

The work in [9] integrates utility-based and control-oriented techniques for the management of energy in hosting centers. Nevertheless, only homogeneous servers, working at the same operating frequency, dedicated to a single application are considered.

In this paper, we propose a utility-based optimization approach extending our previous work [5] by considering the DVFS mechanism and improving the VMs placement, capacity allocation, and load balancing solutions. To evaluate the effectiveness of our joint resource allocation approach, we have performed an in-depth comparison analysis with the solutions presented in [22], [30], [32]. In fact, these techniques have been developed in a coherent framework and have been implemented as part of the IBM Tivoli middleware [30]. In the remaining part of the paper, we will refer to this set of techniques as the alternative solution.

In [32], the reference performance model is presented and validated by real experiments on benchmark applications. The performance model is a closed queuing network model which allows evaluating the performance metric of multitier systems with concurrency limits. The model is evaluated both for single and multiclass systems and is based on the mean value analysis algorithm. In their framework, incoming workload is evenly load balanced among physical servers and the scheduling policy applied at each server is processor sharing (i.e., the proportional assignment scheme is implemented). The number of servers to be adopted at each physical tier is determined by a local search algorithm which moves (homogeneous) servers from one tier to another with the aim at improving bottleneck request response time. The pricing model for serving requests and the admission control policy (based on dynamic programming) are discussed in [22]. Finally in [30], the placement of application on physical servers is presented. The problem of placement is reduced to a variant of the class constrained multiple-knapsack problem and a very efficient heuristic procedure is proposed which is also able to reduce the number of applications starts and stops, while equally balancing the load of physical machines.

3 AUTONOMIC RESOURCE MANAGEMENT

This section provides an overview of our autonomic computing approach for the resource allocation problem in a multitier virtualized environment. The main components of the system are discussed in Section 3.1, whereas the main assumptions of the design of the joint resource allocation policies as well as the performance and power models of the system are presented in Section 3.2.

3.1 Virtualized Service Center Infrastructure

The architecture of the service center under study, shown in Fig. 1, includes a set $I$ of heterogeneous servers which run a Virtual Machine Monitor (VMM), configured in work-conserving mode. Server physical resources are partitioned among multiple VMs, each one hosting a single application (the terms VM and application will be used interchangeably throughout the paper). As in [30], among the many servers’ resources, we consider CPU and memory as representative for the resource allocation problem. The resource allocator exploits the VMM API to dynamically partition CPU capacity and RAM among multiple VMs and their hosted applications. In our prototype environment, we adopt Xen as VMM; however, the approach is independent of the VMM chosen. Multiple instances of the same application can run in parallel on different VMs and each external request can involve the execution of several multitier server applications according to the client/server paradigm. In the system, each server has a single CPU\(^3\) which supports DVFS by varying both its supply voltage and operating frequency from a limited set of values. The adoption of DVFS is very promising since it does not introduce any system overhead, while hibernating and restoring a server require time and energy. Following [24], [27], we adopt full system power models and we assume that the power consumption of a server depends on its operating frequency/voltage and also on the CPU utilization. Moreover, as in [19], [17], [13], the server performance is assumed to be linear in its operating frequency.

The system includes a centralized network controller since the current implementation scales very well with the number of servers and VMs. However, the system could also be implemented with a hierarchical architecture, as in [20], [3], where a first layer controller partitions physical servers among multiple VMs, which are managed by second layer controllers. The main components of the network controller are a monitor, a predictor, an admission controller, and a resource allocator [7] (see Fig. 1). The monitor measures the workload and performance metrics of each application, identifies multiple requests classes, and estimates requests service demands [21]. Request classes are

3. This assumption, made to simplify the autonomic infrastructure model and the paper notation, will be relaxed in Section 6.2.
subsets of requests that are homogeneous with respect to the specific set of VMs required to support their executions, their SLA contracts, and workload profiles (service demands, e.g., CPU time). The predictor forecasts future system load conditions based on historical data [6]. The admission controller decides whether to accept or reject new requests according to the current system workload [10]. Finally, the allocator determines the best system configuration and the assignment of VMs to servers by solving, on multiple time scales, short- and long-term planning problems. This paper focuses on the design of the resource allocator component, while we assume that the incoming workload results from prediction and admission control and service demands are obtained by the monitoring infrastructure, e.g., [21]. Overall, the system serves a set $K$ of request classes. As in [6], [33], for each request class $k \in K$, a linear utility function (see Fig. 2a) specifies the per request revenue (or penalty) $V_k$ incurred when the average end-to-end response time $R_k$, from multiple applications, assumes a given value. The slope of the utility function is $-a_k$ where $a_k = v_k/R_k > 0$ and $R_k$ is the threshold that identifies the revenue/penalty region, that is, if $R_k > R_k$, the SLA is violated and penalties are incurred.

The evaluation of multiclass average response time for a virtualized system is discussed in the next section.

### 3.2 System Performance and Power Models

The service center is modeled by a queuing network composed of a set of multiclass single-server queues and a delay center (see Fig. 2b). Each layer of queues represents the collection of applications supporting requests execution at each tier. The delay center represents the client-based delays, or think time, between the server completion of one request and the arrival of the subsequent request within a session. For each request class $k$, there is a different set of applications $N_k$ which support its execution. User sessions begin with a class $k$ request arriving to the service center from an exogenous source with rate $\lambda_k$. Upon completion, the request either returns to the system as a class $k$ request with probability $\pi_{k,0}$ or it completes with probability $1 - \sum_{l \in K} \pi_{k,l}$. The aggregate rate of arrivals for class $k$ requests, denoted with $\Lambda_k$, is given by the exogenous arrival rates and the transition probabilities: $\Lambda_k = \lambda_k + \sum_{l \neq k} \pi_{k,l} \lambda_l$. Let $\mu_{k,j}$ denote the maximum service rate of a capacity 1 server dedicated to the execution of the application at tier $j$ of request class $k$. 

![Fig. 1. Autonomic computing infrastructure.](image1)

![Fig. 2. Requests utility functions, system performance, and power models.](image2a, image2b, image2c)
We assume that $C_{22k;j}$ parameters are estimated at runtime by the monitoring component (see, e.g., [21]). Moreover, requests at different application tiers within each class and on every server are executed according to the processor sharing scheduling discipline (which is common among webservers and application containers) and service time of class $k$ requests at server $i$ follows a general distribution with mean $\frac{C_i}{1 + \frac{R_{k_i}}{C_{22k;i}}}$, where $C_i$ is the capacity of server $i$ at frequency $h$.

Our objective is to determine:

1. the server to be in active state, introducing for each server $i$ a binary variable $x_i$ equal to 1 if server $i$ is running, 0 otherwise;
2. the frequency of operation of servers’ CPU, given by the binary variable $y_{i,h}$ equal to 1 if server $i$ is working at frequency $h$, 0 otherwise;
3. the set of applications executed by each server, introducing the binary variable $z_{i,k,j}$ equal to 1 if the application tier $j$ for class $k$ request is assigned to server $i$, 0 otherwise;
4. the rate of execution for class $k$ requests at tier $j$ on server $i$, denoted by $\lambda_{i,k,j}$; and
5. the fraction of capacity devoted for executing VM at tier $j$ for request class $k$ at server $i$, denoted by $\phi_{i,k,j}$.

For the sake of clarity, the notation adopted in this paper is summarized in Table 1. Once a capacity assignment has been performed, VMMs guarantee to a given VM and the hosted application to receive as much resource (in terms of CPU capacity and RAM) as it has been assigned to, regardless of the load imposed by other applications. For example, if two VMs have been assigned the same fraction of capacity $\phi_{i,k,j} = 0.5$, then they are guaranteed to receive 50 percent of the CPU each. Since the VMM has been configured to support work-conserving mode (see [14]), if one of the two aforementioned VMs is blocked for an I/O operation (or simply because no requests are running in the system), then the other one can consume the entire CPU. The service discipline implemented by a VMM configured in work-conserving mode can be modeled as first approximation by the Generalized Processor Sharing (GPS) scheduling [37]. Under GPS, the capacity of server $i$ devoted to class $k$ request for application tier $j$ at time $t$ (if any), when the server is working at frequency $h$, is

$$
C_{i,h} \frac{\phi_{i,k,j}}{K(t) \in K_i(t) \lambda_{i,k,j} t^{\frac{1}{h}}},
$$

where $K_i(t)$ is the set of classes with waiting requests on server $i$ at time $t$. To evaluate the requests response time, we adopt the approximation discussed in [37] and widely used in the literature [2], [35]. Each multiclass single-server queue associated with a server application is decomposed into multiple independent single-class single-server $M/G/1$.
queues with capacity greater than or equal to \( C_{i,h}\phi_{i,k,j} \). The response times evaluated in the isolated per-class queues are upper bounds of the corresponding measures in the original system. \( R_{i,k,j} \), i.e., the average response time for the execution of the application at tier \( j \) of class \( k \) request at server \( i \) working at frequency \( h \) can be evaluated as

\[
R_{i,k,j} = \frac{1}{C_{i,h}\mu_{i,k,j} - \lambda_{i,k,j}}. \tag{1}
\]

The average response time for class \( k \) requests is the sum of the average response times at each tier computed over all servers:

\[
R_k = \frac{1}{\lambda_k} \left( \sum_{i \in I} \lambda_{i,k,j} R_{i,k,j} \right). \tag{2}
\]

System performance is evaluated by adopting analytical models as in [22], [2], [32], [35]. There is a tradeoff between the accuracy of the model and the time required to estimate system performance which have to be evaluated with strict time constraints. Accurate performance models have been provided in the literature for web systems, e.g., [26], [15]. However, the high complexity for analyzing even small size models has prevented us from using them here. Note also that the approximation provided in [37] is asymptotically correct for high loads. In [4], we have shown that the average percentage error introduced by (1) is around 30 percent for high utilization of the physical server (above 85 percent). Note that, in the current practice of server consolidation, the aim is to increase the CPUs utilization and 80 percent average utilization has been reached on x86 server farms [34] (traditionally, mainframe environments are characterized by higher utilization, around 90 percent). We performed an extensive analysis of the accuracy which can be achieved by (1). Results have shown that the accuracy depends on the application, and in some cases also on the application mixes. For low utilization, the performance estimated by (1) for a single VM could be conservative with respect to the values obtained in real systems. However, usually under light load, different resource allocation strategies perform equally well, while under heavy load, the resource allocation problem is more severe (see, e.g., [3], [38]). The energy related cost is computed using the full system power models adopted in [24], [27], which allow estimating servers energy consumption within a 10 percent accuracy. A server in low power sleep state still consumes energy (\( \varsigma \), cost parameter) [19], [20], while when it is running, its dynamic operating cost depends on a constant base cost \( \tau_i + p_{i,h} \), which is a function of the operating frequency, and on an additional cost component which varies linearly with the CPU utilization \( U_i \) with slope \( b_{i,h} \)[24] (see Fig. 2c). Furthermore, as in [19], [30], we consider the switching cost \( c_{s_i} \) which includes the power consumption costs incurred when starting up the server from its low power sleep state and, possibly, the revenue lost when the computer is unavailable to perform any useful service for some duration. Finally, to limit the number of VM movements in the system, we add the movement cost \( c_m \), which avoids system instability and minimizes the total number of applications starts and stops [19], [30].

4 Long-Term Optimization Problem and Solution Technique

In this section, we present the optimization model and the corresponding solution procedure for the long-term problem. The objective is to maximize the difference between revenues from SLA contracts and the costs associated with servers switching and moving VMs for the next control time horizon \( T \). The revenue for class \( k \) requests is

\[
V_k \Lambda_k T = (v_k - a_k R_k) \Lambda_k T.
\]

The total cost is given by the cost of servers in lower power sleep state, the cost of using servers according to their operating frequency, the penalties incurred when switching servers from the low power sleep to the active state, and when moving VMs. Let us denote with \( \tau_i \) and \( \varsigma_{i,k,j} \) the values of the variables \( x_i \) and \( z_{i,k,j} \) in the solution of the previous control time period. Since the utilization of server \( i \) when it is working at frequency \( h \) is

\[
U_i = \sum_{k \in K, j \in N_i} \frac{\lambda_{i,k,j}}{\lambda_k T},
\]

the objective function is

\[
\begin{align*}
& \max \left[ \sum_{i \in I} \left( \sum_{k \in K} \left( -a_k \sum_{j \in N_i} \left( \sum_{h \in H_i} C_{i,h} y_{i,h} \right) \mu_{i,k,j} \phi_{i,k,j} - \lambda_{i,k,j} \right) \right. \right. \\
& \quad - \left. \sum_{i \in I, k \in K, j \in N_i} \left( p_{i,h} y_{i,h} + b_{i,h} y_{i,h} \sum_{j \in N_i} \sum_{k \in K} \lambda_{i,k,j} \mu_{i,k,j} \right) \right) \\
& \quad + \sum_{k \in K} v_k \Lambda_k T - \sum_{i \in I} c_{s_i} \max(0, x_i - \tau_i) \\
& \quad - \sum_{i \in I, k \in K, j \in N_i} \frac{c_m}{T} \max(0, z_{i,k,j} - \varsigma_{i,k,j}) \right].
\end{align*}
\]

Note that the terms \( \sum_{k \in K} v_k \Lambda_k T \) and \( \sum_{i \in I} \tau_i T \) can be dropped in the objective function since they are independent of the decision variables. The long-term optimization problem (solved every half an hour as described in Section 1) can be formulated as follows:

\[
(P1) \quad \max \sum_{i \in I, k \in K, j \in N_i} \left( \frac{-a_k \lambda_{i,k,j}}{\sum_{h \in H_i} C_{i,h} y_{i,h} \mu_{i,k,j}} \phi_{i,k,j} - \lambda_{i,k,j} \right) \\
- \sum_{i \in I, k \in K, j \in N_i} \left( p_{i,h} y_{i,h} + b_{i,h} y_{i,h} \sum_{j \in N_i} \sum_{k \in K} \lambda_{i,k,j} \mu_{i,k,j} \right) \\
- \sum_{i \in I} c_{s_i} \max(0, x_i - \tau_i) \\
- \sum_{i \in I, k \in K, j \in N_i} \frac{c_m}{T} \max(0, z_{i,k,j} - \varsigma_{i,k,j})
\]

subject to

\[
\sum_{i \in I} \lambda_{i,k,j} = \Lambda_k = \Lambda_{k,j} \quad \forall k \in K, j \in N_k, \tag{4}
\]

\[
\sum_{k \in K, j \in N_k} \phi_{i,k,j} \leq 1 \quad \forall i \in I, \tag{5}
\]

\[
\sum_{h \in H_i} y_{i,h} = x_i \quad \forall i \in I, \tag{6}
\]

\[
z_{i,k,j} \leq x_i \quad \forall i \in I, k \in K, j \in N_k, \tag{7}
\]

\[
\lambda_{i,k,j} \leq \Lambda_{k,z_{i,k,j}} \quad \forall i \in I, k \in K, j \in N_k. \tag{8}
\]
\begin{equation}
\lambda_{i,k,j} < \left( \sum_{h \in H} C_{i,h} y_{i,h} \right) \mu_{k,j} \phi_{i,k,j} \quad \forall i \in I, k \in K, j \in N_k, \tag{9}
\end{equation}

\begin{equation}
\sum_{j \in N_k} RAM_{k,j} z_{i,k,j} \leq \text{RAM}_i \quad \forall i \in I, k \in K,
\lambda_{i,k,j}, \phi_{i,k,j} \geq 0, \quad x_i, y_{i,h}, z_{i,k,j} \in \{0,1\} \quad \forall i \in I, k \in K, j \in N_k.
\end{equation}

Constraints (4) ensure that the traffic assigned to individual servers and for every application tier equals the overall load predicted for class \( k \) jobs. Note that, for a given request class \( k \), the overall load \( \lambda_k \) is the same at every tier. Constraints (5) express the bounds for the VMs capacity allocation (i.e., at most 100 percent of the server capacity can be used). Constraints (6) guarantee that, if the server \( i \) is in active state, then exactly one frequency is selected in the set \( H_i \), i.e., only one variable \( y_{i,h} \) is set to 1. Hence, the extra cost of a server working at frequency \( h \) and its corresponding capacity can be expressed by \( \sum_{h \in H} b_{i,h} y_{i,h} \) and \( \sum_{h \in H} C_{i,h} b_{i,h} \), respectively. Constraints (7) state that application tiers can only be assigned to servers in active state. Constraints (8) allow running requests at a server only if the corresponding application tier has been assigned to it. Constraints (9) guarantee that resources are not saturated. Finally, constraints (10) ensure that the memory available at a given server is sufficient to support all the VMs assigned to it.

The model (P1) is a mixed integer nonlinear programming problem known to be difficult to solve for large number of variables. Even if we fix the set of servers in active state \((x = (x_{i,j} \in \mathbb{E}) \) with relative frequencies \((y = (y_{i,h} \in \mathbb{R}) \) and we assign the applications on each server \((z = (z_{i,k,j} \in \mathbb{R}) \) \) are obtained using a greedy heuristic, and 2) the current solution is improved using a local search procedure which is defined in the space of variables \( x, y, \) and \( z \) while updates the \( \lambda \) and \( \phi \) values of a limited number of servers involved. The moves adopted in the exploration of the next neighborhood constitute the procedure \textit{Apply_Best_Move()} , discussed in Section 4.3. When the exploration of the first neighborhood does not improve the current solution (step 6), i.e., a local optimum is found, the FPI is executed again (step 7) in order to determine a local minimum with respect to the capacity allocation and the load balancing problems. Thus, the second neighborhood is defined in the space of variables \( \phi \) and \( \lambda \). The local search is iteratively restarted until a local optimum is found with respect to both neighborhoods (step 9).

The obtained final local optimum solution is therefore robust to the perturbation of each individual decision variable (integer or continuum) of the optimization problem.

### 4.1 Building an Initial Solution

We have considered two versions of the greedy procedure that generates the initial solution. Both implementations are based on the \textit{greedyCore()} procedure. The basic idea is to assign application tiers to servers while guaranteeing that CPU utilization of servers is lower than a given threshold \( U \).

The value of \( U \) has been selected in the range 50-60 percent according to literature proposals (e.g., [1], [10], [28]). Let \( \overline{C}_i \) and \( \overline{b}_i \) denote, respectively, the values of \( C_{i,h} \) and \( p_{i,h} + b_{i,h} \), for \( y_{i,h} = 1 \). Moreover, let \( \Lambda_{\text{res}} \) be the bidimensional matrix \((\Lambda_{\text{res}}_{i,j})_{i \in K, j \in \mathbb{R}} \) representing the remaining workload to be allocated. In \textit{greedyCore()}, the procedure \textit{Algorithm 4}, the application tiers are initially ordered according to nondecreasing value \( W_{k,j} = (\Lambda_{\text{res}}_{i,j})_{i \in K, j \in \mathbb{R}} \) and servers are ordered according to nondecreasing value of the ratio \( \overline{C}_i / \overline{b}_i \). Then, the procedure iteratively selects a (partially allocated) application tier and assigns a class \( k \) request at tier \( j \) to a server \( i \). The process is repeated until \( \Lambda_{\text{res}}_{i,j} \) is completely allocated while guaranteeing that servers utilization is at most \( U \) and according also to the RAM constraints (10). When a server is turned in active state (steps 8-11), it is set to the maximum frequency. If it is possible, the workload \( \Lambda_{\text{res}}_{i,j} \) is allocated completely on server \( i \) and after updating the server utilization, the next tier is selected (steps 16-19). On the contrary, if \( \Lambda_{\text{res}}_{i,j} \) cannot be allocated completely on the
single server $i$ (step 12), then the workload is partially assigned to the server guaranteeing $U_i$ maximum utilization (steps 12-15). The remaining load (computed in step 15) is allocated on subsequent servers.

The complexity of the greedyCore() is $O(\max\{|I| \cdot \sum_{k \in K} |N_k|, \sum_{k \in K} |N_k| \cdot \log(\sum_{k \in K} |N_k|), |I| \cdot \log(|I|))$, where the first term is the number of $\lambda$ assignments performed under the worst case scenario: At each iteration, the first server with enough RAM and residual computing capacity is the last one in the servers ordering, while $O(\sum_{k \in K} |N_k| \cdot \log(\sum_{k \in K} |N_k|))$ and $O(|I| \cdot \log(|I|))$ are the sorting time complexity of the application tiers and servers, respectively.

To build a feasible solution, the first version of the procedure (Algorithm 2) starts from scratch: all servers are assumed to be in lower power sleep state and the remaining workload to be allocated is equal to the workload prediction for the next time horizon $\lambda^{\text{res}} = \{\lambda_{kj}\}_{k \in K, j \in N_k}$. The second version (Algorithm 3) tries to improve the solution obtained in the previous control time horizon according to the new workload forecast: the current solution is used as the input to the greedyCore() procedure, while the load balancing is computed proportionally to the previous solution (step 4).

When in this way a VM is saturated (step 11), we first try to assign the corresponding VM to the server guaranteeing the maximum utilization of single server $i$ (step 12), then the workload is partially assigned to the server guaranteeing $U_i$ maximum utilization (steps 12-15). The remaining load (computed in step 15) is allocated on subsequent servers.

To build a feasible solution, the first version of the procedure (Algorithm 2) starts from scratch: all servers are assumed to be in lower power sleep state and the remaining workload to be allocated is equal to the workload prediction for the next time horizon $\lambda^{\text{res}} = \{\lambda_{kj}\}_{k \in K, j \in N_k}$. The second version (Algorithm 3) tries to improve the solution obtained in the previous control time horizon according to the new workload forecast: the current solution is used as the input to the greedyCore() procedure, while the load balancing is computed proportionally to the previous solution (step 4).

When in this way a VM is saturated (step 11), we first try to set the physical server to the maximum frequency (steps 12-15). The remaining load (computed in step 15) is allocated on subsequent servers.

To build a feasible solution, the first version of the procedure (Algorithm 2) starts from scratch: all servers are assumed to be in lower power sleep state and the remaining workload to be allocated is equal to the workload prediction for the next time horizon $\lambda^{\text{res}} = \{\lambda_{kj}\}_{k \in K, j \in N_k}$. The second version (Algorithm 3) tries to improve the solution obtained in the previous control time horizon according to the new workload forecast: the current solution is used as the input to the greedyCore() procedure, while the load balancing is computed proportionally to the previous solution (step 4).

When in this way a VM is saturated (step 11), we first try to set the physical server to the maximum frequency (steps 12-15). The remaining load (computed in step 15) is allocated on subsequent servers.

Algorithm 4. greedyCore(): Application to server allocation

```plaintext
input : $\mathcal{C}, \mathcal{R}, x, \lambda, \phi, \lambda^{\text{res}}, \text{ram}$
output : $x, \lambda, \phi, y, z$
1 $W_{kj} \leftarrow M_{kj}^{\text{res}}, \sum_{j' < j} \frac{\lambda_{kj}}{|I|}, N'J = \{(k, j) | k \in K, j \in N_j\};$
2 $(k, j) \leftarrow \arg\max\{W_{kj} | (k, j) \in N'J\};$
3 while $N'J \neq \emptyset$ do
4 $i \leftarrow \arg\min\{\frac{\lambda_{kj}}{|I|} | i \in I \text{ and } \sum_{j' \in J \cap N_i} \phi_{kj} > 0 \text{ and } \text{ram}_{i} \geq \text{RAM}_{kj} \}$
5 if $\phi_{kj} = 0$ then
6 $\phi_{kj} \leftarrow 1;
7 \text{ram}_{i} \leftarrow \text{ram}_{i} - \text{RAM}_{kj};$
8 end
9 if $x_i = 0$ then
10 $x_i \leftarrow 1;
11 y_{i,h} \leftarrow 1 \text{ s.t. } h \text{ index corresponds to the maximum frequency server } i;$
12 end
13 end
14 $\lambda^{\text{res}} \leftarrow \{\lambda_{kj} \}_{k \in K, j \in N_k};$
15 greedyCore($\mathcal{C}, \mathcal{R}, x, \lambda, \phi, \lambda^{\text{res}}, \text{ram}, y, z$);
```

Algorithm 3. Application to server allocation from scratch using previous solution

$$
\lambda^{\text{res}} \leftarrow \lambda_{kj}, \forall i, k \in K, j \in N_j \text{ s.t. } \bar{A}_{kj} = 0; \\
\lambda_{kj} \leftarrow 0, \forall i, k \in K, j \in N_j \text{ s.t. } \bar{A}_{kj} = 0; \\
\lambda_{kj}^{\text{res}} \leftarrow 0, \forall i, k \in K, j \in N_j \text{ s.t. } \bar{A}_{kj} > 0; \\
\lambda_{kj}^{\text{res}} \leftarrow \lambda_{kj}^{\text{res}} - \lambda_{kj}, \forall i, k \in K, j \in N_j \text{ s.t. } \bar{A}_{kj} > 0; \\
x_i \leftarrow x_i, \forall i \in I; \\
y_{i,h} \leftarrow y_{i,h}, \forall i \in \text{H, h}; \\
\phi_{kj} \leftarrow \phi_{kj}, \forall i, k \in K, j \in N_j; \\
\bar{C}_i \leftarrow \sum_{h \in \text{H}} C_{i,h} \text{;} \\
\bar{c}_i \leftarrow \sum_{h \in \text{H}} c_{i,h} \text{;} \\
\text{for } i \in I, k \in K, j \in N_j \text{ do}
$$

if $\lambda_{kj} \geq \bar{C}_{i,k,j} \phi_{kj}$ then

update $y_{i,h}$, s.t. server $i$ is set to the maximum frequency;
update $\bar{C}_i$;
end

if $\lambda_{kj}^{\text{res}} \geq \bar{C}_{i,k,j} \phi_{kj}$ then

$\lambda_{kj}^{\text{res}} \leftarrow \lambda_{kj}^{\text{res}} - \bar{C}_{i,k,j} \phi_{kj}, \forall i, k \in K, j \in N_j \text{;} \\
\lambda_{kj}^{\text{res}} \leftarrow \bar{U}_{i,k} \phi_{kj}, \forall i, k \in K, j \in N_j ;$
end

greedyCore($\mathcal{C}, \mathcal{R}, x, \lambda, \phi, \lambda^{\text{res}}, \text{ram}, y, z$);

4.2 The Capacity Allocation and Load Balancing Problems

At step 2 of Algorithm 1, once the application tiers and the operating frequencies are assigned to servers, the capacity allocation at each server and the load balancing can be improved. Let $I_{\text{active}} = \{i | x_i = 1\}$ denote the set of servers in active state, $\bar{C}_i = \sum_{h \in \text{H}} C_{i,h} y_{i,h}$ and $\bar{b}_i = \sum_{h \in \text{H}} b_{i,h} y_{i,h}$. The joint capacity allocation and load balancing problem can be modeled as follows:

(P2) \[
\min_{k \in K} \sum_{j \in N_j} \left( a_k \sum_{i \in I_{\text{active}}} \frac{\lambda_{kj}}{\bar{C}_j + b_k} + \frac{\bar{b}_k}{\bar{C}_i} \sum_{j \in N_j} \lambda_{kj} \right) \\
\text{subject to } \sum_{k \in K} \lambda_{kj} = \lambda_{kj}^{\text{res}} \forall k \in K, j \in N_j, \\
\sum_{k \in K} \phi_{kj} \leq 1 \forall i \in I_{\text{active}}, \\
\lambda_{kj} \leq \bar{z}_{kj}, \lambda_{kj} \leq \bar{C}_j \phi_{kj}, \lambda_{kj} \geq 0 \forall i \in I_{\text{active}}, k \in K, j \in N_j,
\] where the decision variables are $\lambda_{kj}$ and $\phi_{kj}$, since $z_{kj}$ are held fixed and depend on the current application tiers to servers assignment. Note that the goal is to minimize the weighted average response time of request classes. We have applied an FPI technique, which iteratively identifies the optimal value of a set of variables ($\lambda$ or $\phi$), while the value of the other one (alternatively $\phi$ or $\lambda$) is held fixed.

4.2.1 The Load Balancing Subproblem

Let the capacity allocation variables $\phi_{kj}$ be fixed to the values $\bar{\phi}_{kj}$, then the load balancing subproblem is as follows:

(P3) \[
\min_{k \in K, j \in N_j} \left( a_k \frac{\lambda_{kj}}{\bar{C}_j \phi_{kj}} - \frac{\bar{b}_k}{\bar{C}_i} \right) \\
\text{subject to } \sum_{j \in N_j} \lambda_{kj} = \lambda_{kj}^{\text{res}}, \\
\text{and } \lambda_{kj} \geq 0 \forall k \in K, j \in N_j.
\]
where $\lambda_{i,k,j}$ are the only decision variables. The goal is to minimize the weighted average response times at every tier and every server. Problem (P3) can be solved using its separable structure: it suffices to solve the $\sum_{k \in K} |N_k|$ load balancing subproblems (one for every tier of each class) independently. The objective function is convex since the Hessian is given by

$$\text{Diag} \left( \frac{2m_i \overline{C}_{i,j} \overline{P}_{i,j} - \lambda_{i,k,j}}{\left( \overline{C}_{i,j} \overline{P}_{i,j} - \lambda_{i,k,j} \right)^2} \right)$$

and its eigenvalues are positives. Ignoring $k$ and $j$ indexes, each of the subproblem in which problem (P3) can be separated is as follows:

\[(P4) \quad \min_{\{i \in I_{\text{active}}\}} \sum_{i \in I_{\text{active}}} \lambda_i \left( \frac{a_k}{\Theta_i - \lambda_i} + \frac{b_i}{\overline{C}_{i,j}} \right), \quad \sum_{i \in I_{\text{active}}} \lambda_i = \Lambda, \quad 0 \leq \lambda_i < \Theta_i, \quad \forall i \in I_{\text{active}}.\]

If a tier is not allocated to server $i$, then the capacity $\Theta_i$ is set to 0, otherwise $\Theta_i = \overline{C}_{i,j} \overline{P}_{i,j}$. The constraints $\lambda_i \leq \lambda_k z_{i,k,j}$ of problem (P3) are omitted under the assumption that a feasible assignment of application tiers to servers is performed. The solution is obtained through the following theorem:

**Theorem 1.** In the optimal solution of problem (P4), let $I_{\text{active}} \subseteq I_{\text{active}}$ be the set of indexes $i$ s.t. $\lambda_i > 0$ and let $\overline{7}$ be an arbitrarily chosen index in $I_{\text{active}}$. Then,

$$\lambda_i = \Theta_i - \frac{\sqrt{a \Theta_i (\Theta_i - \lambda_i)}}{\sqrt{a \Theta_i + \left( \frac{\overline{C}_{i,j}}{\overline{C}_{i,j}} - \frac{\overline{C}_{i,j}}{\overline{C}_{i,j}} \right) (\Theta_i - \lambda_i)^2}} \quad \forall i \in I_{\text{active}} - \{\overline{7}\}, \quad (11)$$

where $\lambda_i$ is such that

$$\sum_{i \in I_{\text{active}}} \Theta_i - \frac{\sqrt{a \Theta_i (\Theta_i - \lambda_i)}}{\sqrt{a \Theta_i + \left( \frac{\overline{C}_{i,j}}{\overline{C}_{i,j}} - \frac{\overline{C}_{i,j}}{\overline{C}_{i,j}} \right) (\Theta_i - \lambda_i)^2}} = \Lambda. \quad (12)$$

**Proof.** See Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputer.org/10.1109/TSC.2010.42.

The meaning of Theorem 1 is that, fixed an arbitrary active server $\overline{7}$, we can easily evaluate the corresponding arrival rate $\lambda_{\overline{7}}$ from the characteristics of the servers supporting application instances working in parallel (i.e., $\Theta_{\overline{7}}, b_{\overline{7}},$ and $\overline{C}_{\overline{7}}$). Therefore, by a closed expression (11), all the load balancing variables $\lambda_i$ corresponding to the other servers are computed from $\lambda_{\overline{7}}$. In the worst case, all servers are in active state; thus, problem (P4) can be solved by performing $O(|I|^2)$ operations and the complexity of (P3) is $O(|I|^2 \sum_{k \in K} |N_k|)$. Equation (14) has been solved by using Newton’s method.

### 4.2.2 The Capacity Allocation Subproblem

Let the load balancing variables $\lambda_{i,k,j}$ be fixed to the values $\overline{\lambda}_{i,k,j}$, then the capacity allocation problem is as follows:

\[(P5) \quad \min \sum_{k \in K} \sum_{i \in I_{\text{active}}} \phi_{i,k,j} - \lambda_{i,k,j}, \quad \sum_{k \in K} \phi_{i,k,j} \leq 1, \quad \forall i \in I_{\text{active}}, \quad \overline{\lambda}_{i,k,j} > \overline{\lambda}_{i,k,j}, \quad \forall k \in K, j \in N_k.

where $\lambda_{i,k,j}$, $\overline{\lambda}_{i,k,j}$, $\phi_{i,k,j}$ are the only decision variables. Again, the goal is to minimize the weighted average response times at every tier for every server. Problem (P5) is separable and the $|I|$ capacity allocation subproblems, one for each server, can be solved independently. The objective function is convex since the Hessian, given by

$$\text{Diag} \left( \frac{2 \overline{a} \overline{C}_{i,j}^2 \overline{P}_{i,j}^2 - \overline{\lambda}_{i,k,j}^2}{\left( \overline{C}_{i,j} \overline{P}_{i,j} - \overline{\lambda}_{i,k,j} \right)^2} \right),$$

has positive eigenvalues. By omitting $i$ indexes, each of the subproblem in which problem (P5) can be separated is as follows:

\[(P6) \quad \min \sum_{k \in K} \sum_{j \in N_k} \lambda_{i,k,j} \overline{C}_{i,j} \overline{P}_{i,j} - \overline{\lambda}_{i,k,j}, \quad \sum_{k \in K} \phi_{i,k,j} \leq 1, \quad \phi_{i,k,j} > \overline{\lambda}_{i,k,j}, \quad \forall k \in K, j \in N_k.

The solution is obtained through the following theorem:

**Theorem 2.** In the optimal solution of problem (P6), let $\overline{N_i,j}$ be the set of indexes $(k,j)$ s.t. $\overline{\lambda}_{i,k,j} > 0$, and let $(\overline{k}, \overline{j})$ an arbitrarily element in $\overline{N_i,j}$. Then,

$$\phi_{\overline{k}, \overline{j}} = \frac{1}{\overline{C}_{\overline{k}, \overline{j}}} \left( \frac{\overline{a}_k \overline{M}_{\overline{k}, \overline{j}} \overline{\lambda}_{i,k,j}}{\overline{a}_k \overline{M}_{\overline{k}, \overline{j}} - \overline{\lambda}_{i,k,j}} \right)$$

\[\forall (k,j) \in \overline{N_i,j} - \{(\overline{k}, \overline{j})\},

where

$$\phi_{\overline{k}, \overline{j}} = \frac{\overline{\lambda}_{i,k,j}}{\overline{C}_{i,j} \overline{P}_{i,j}} + \frac{1 - \sum_{k \in K} \sum_{j \in N_k} \overline{\lambda}_{i,k,j}}{\sum_{k \in K} \sum_{j \in N_k} \overline{a}_k \overline{M}_{\overline{k}, \overline{j}}}. \quad (16)$$

**Proof.** See Appendix A, which is available in the online supplemental material.

As for Theorem 1, the significance of the previous theorem is that, fixed an application tier $\overline{j}$ for class $\overline{k}$ request, we can evaluate by a closed expression both, the capacity allocation variables corresponding to application tier $\overline{j}$ for class $\overline{k}$ request, $\overline{\phi}_{\overline{k}, \overline{j}}$, and therefore the ones corresponding to the other applications $\phi_{i,k,j}$. The capacity allocation $\overline{\phi}_{\overline{k}, \overline{j}}$ is computed from the rate of execution, the maximum service rate for all class at different tiers, and from the utility function slopes (i.e., $\overline{\lambda}_{i,k,j}, \mu_{i,k,j}, a_k$).
Since each of the $|I|$ subproblems (P6) has complexity $O(|I| \sum_{k \in K} |N_k|)$, problem (P5) can be solved by performing $O(|I| \cdot \sum_{k \in K} |N_k|)$ operations.

The FPI iteratively solves the load balancing and capacity allocation problems. These problems are neither concave nor convex (see [5]); therefore, the procedure is not guaranteed to converge to a global optimum, however it is asymptotically convergent. In fact, since at each step the current $\lambda$ and $\phi$ assignment is improved, the FPI will find a local optimal solution. Experimental results show that the number of iterations required for the FPI to converge within the precision $10^{-5}$ is typically between 6 and 20.

### 4.3 The Neighborhood Exploration

At step 2 of Algorithm 1, the result of the FPI procedure is a feasible solution $s$ of problem (P1) which is a local optimum with respect to the decision variables $\phi$ and $\lambda$. To further improve $s$, we apply a local search procedure which identifies a local optimum with respect to the variable $x$, $y$, and $z$, while the variables $\phi$ and $\lambda$ are updated in a greedy way. The neighborhood $N(s)$ in the local search procedure is given by all those solutions that can be obtained by applying four different kind of moves: setting a server in active or low power sleep state, scaling the server frequency, and reallocating tiers to servers. The procedure $\text{Apply}_{\text{Best}}_{\text{Move}}(s)$ identifies the best solution in $N(s)$. In the following, we illustrate the moves in the local search.

**Set servers in low power sleep state.** All servers $i \in I_{\text{active}}$ whose utilization $U_i$ is in the interval $[\min U_i, \alpha \min U_i]$, with $\alpha$ a constant experimentally set between 1.1 and 1.2, are considered lightly loaded and are candidates to be switched into low power sleep state. The load of the server to be switched into low power sleep state, say $i$, is allocated on the remaining servers proportionally to their spare capacity: the spare capacity for tier $j$ of class $k$ job at server $i$ is $S_{i,j,k} = \frac{C_i - \phi_{i,j,k} - \lambda_{i,j,k}}{C_i}$. Let $S_k = \sum_{i \in I_{\text{active}}} S_{i,j,k}$ be the overall spare capacity available at remaining servers in active state for tier $j$ of class $k$. The load at server $i$ is assigned to remaining servers setting $\lambda_{i,j,k} = \lambda_{i,j,k} + S_{i,j,k} S_k$. Note that, this suboptimal load balancing is improved when the local search reaches a local optimum by running the FPI again (step 7 of Algorithm 1).

The neighborhood exploration has complexity $O(|I|^2 \sum_{k \in K} |N_k|)$ in the worst case scenario where all the servers are in active state and have the same utilization and every application tier is allocated on every server.

**Set servers in active state.** All servers $i \in I_{\text{active}}$ with $U_i \in [\beta \max U_i, \max U_i]$, where $\beta$ is a constant experimentally set between 0.9 and 0.95, are considered as bottleneck servers. To alleviate the load at a bottleneck server, say $i_1$, a server in low power sleep state, say $i_2$, with $\frac{\text{RAM}_{i_2}}{\text{RAM}_{i_1}}$ greater or equal to the memory allocated on server $i_1$, is set in active state. The server $i_2$ applies the same capacity allocation adopted by $i_1$ and runs at the maximum frequency. The optimal load balancing is identified by solving an instance of problem (P3), limited to the servers $i_1$ and $i_2$. The neighborhood exploration complexity is $O(|I|^2 \sum_{k \in K} |N_k|)$, when the system is running with $|I|/2$ servers in active state with almost the same utilization.

**Frequency scaling.** This move updates the servers’ frequency by one step. In particular, for bottleneck servers, the frequency is increased, while for lightly loaded servers, the frequency is decreased (only if constraints (9) are not violated). The rationale is that the increase in the cost of using resources can be balanced improving system performance, while the energy savings can be counterbalanced by the worsening of response times.

**Reallocation of application tiers to servers.** This move does not modify the server configurations of the service center but tries to reallocate application tiers to servers already in active state in order to find more profitable solutions. The aim is to look for application tiers which can be allocated on a different set of servers. The move is based on the following:

**Property 1.** For a given iteration, if the FPI procedure sets the value of a variable $\lambda_{i,j,k}$ to 0, then in the final solution, $\lambda_{i,j,k} = 0$.

**Proof.** See Appendix A, which is available in the online supplemental material.

As a consequence of Property 1, the FPI procedure can not allocate new application tiers on servers. If an application tier is deallocated from a server $i$, the corresponding $z$ variable is set to zero and a tier which was not allowed to be executed on server $i$, because of a memory constraint, can now be allocated on it. The idea is to make use of these new degrees of freedom and try to allocate on such servers new application tiers. The basic steps of the move are:

1. identify a destination server $i_1$,
2. identify a candidate tier $(k_1,j_1)$ to be moved,
3. identify a source server $i_2$,
4. create spare capacity on $i_1$, and
5. balance the load of application tier $(k_1,j_1)$ between $i_1$ and $i_2$.

We consider as destination servers all the servers $i \in I_{\text{active}}$ with enough memory capacity, such that $U_i < U$, where $U$ is again in the range 50-60 percent. In fact, allocating another application tier on an overutilized server will unlikely produce more profits. The search stops when an improvement is identified. A source server $i_2$ is characterized by $U_i > U$. Since, in the optimal solution of problem (P5), the sum of $\phi$ variables on each server equals 1 (see proof of Theorem 2), the spare capacity required for allocating $(k_1,j_1)$ is created by setting $\phi_{i_1,j_1} = \frac{U_{i_1}}{U_{i_2}}$. For all application tiers currently executed on server $i_2$, i.e., guaranteeing $U_i < U$. Accordingly, we set $\phi_{i_2,j_1} = 1 - \sum_{k \in K_i} \sum_{j \in J_1} \phi_{i_2,j,k}$ where $K_i$ is the set of request classes currently executed at server $i_2$. The capacity available at server $i_2$ for the new application tier is then $C_{i_2} - \sum_{k \in K_i} \phi_{i_2,j,k}$. Finally, the optimal load balancing between servers $i_1$ and $i_2$ is computed by solving the corresponding restricted instance of problem (P4). The neighborhood exploration complexity is $O(|I|^2 \sum_{k \in K} |N_k|)$ in the worst case scenario where $|I|$ servers are in active state, $|I|/2$ servers have an
utilization less than $U$, and the memory requirements allow every application tier to be allocated on every server.\footnote{The new values computed for $\lambda$ and $\phi$ variables in the local search do not necessarily correspond to the optimal solution which could be better approximated by running the $FPI()$ procedure. We cannot run the $FPI$ procedure for every neighborhood candidate solution $s' \in N(s)$, because it is too time consuming. However, after the first execution of the $FPI()$, we can reasonably assume that the optimal load balancing and capacity allocation solutions are only perturbed by switching servers, changing server frequency, or by reallocating applications. Hence, the candidate solution $s'$ is reasonably close to its corresponding local optimum. This hypothesis has been confirmed by computational experiments. Moreover, when the local search stops at a local optimum, we do execute (see step 7 in Algorithm 1) the $FPI()$ procedure. In this way, we try to escape from the local maximum by optimally updating the $\lambda$ and $\phi$ variables, instead of approximating their values, as usually done in the neighborhood exploration.}

## 5 Short-Term Optimization Problem and Solution Technique

As discussed in Section 1, the autonomic reconfiguration of the system is performed with different time scales. The short-term planning, implemented every 5 minutes, considers only load balancing, capacity allocation, and frequency scaling, since they do not add a significant overhead in the physical system. The short-term optimization problem can be obtained from (P1) by considering as decision variables $\lambda$, $\phi$, and $y$, and fixing the remaining ones. The short-term solution technique implements only the while loop at step 4 of Algorithm 1 and considers only the frequency scaling move in $Apply_{Best\ Move}()$ procedure.

## 6 Numerical Analysis

The resource management algorithms proposed have been evaluated for a variety of system and workload configurations. Section 6.1 presents a cost-benefit evaluation of our solution compared with the top-performing state-of-the-art techniques [22], [30], [32]. Section 6.2 shows the results of the application of our resource allocation policies in a real prototype environment. An evaluation of the energy savings which can be achieved with respect to a static allocation is reported in Section 6.3. The sensitivity of our approach with respect to the prediction of the incoming workload is discussed in Section 6.4. Finally, the scalability analysis of our algorithms is reported in Appendix B, which is available in the online supplemental material.

### 6.1 Comparison with the Alternative Solution

In this section, we compare the long-term algorithm proposed with the alternative solution discussed in Section 2 for realistic workloads created from a trace of requests relative to the website of a large university in Italy. The real system includes almost 100 servers and the trace contains the number of sessions, on a per-hour basis, over a one year period (from 1 January 2006 to 31 December 2006). Realistic workloads are built assuming that the request arrivals follow nonhomogeneous Poisson processes with rates changing every hour according to the trace. From these logs, we have extracted 10 requests classes corresponding to the days with the highest workloads experienced during the year. In particular, two reference case studies have been built. In case 1, a synthetic workload including the 10 requests has been considered (see Fig. 3, where the number of concurrent users per hour is reported). The workload follows a bimodal distribution with two peaks around 11.00 and 16.00. In case 2, five request classes are shifted by 12 hours in order to obtain a more homogeneous overall incoming workload with lower peaks. The new workloads have consequently more heterogeneous profiles, with peaks in different time periods, thus offering more opportunities for resource sharing. Note that, this is representative for a service provider with customers geographically distributed across different time zones.

The comparison between our solution and the alternative one is based on simulation using a log-normal distribution for service times, with coefficient of variation equal to 4 (i.e., the standard deviation is four times the average service time). The simulation allows validating systems with up to 10 servers distributed on two tiers mainly by the time required to simulate the GPS scheduling policy. The results have been obtained with Anylogic 6.0 simulator (see Appendix B, which is available in the online supplemental material, for the details on the experimental settings, e.g., the number of tiers, maximum service rates, etc., we have considered in this paper).

![Fig. 3. Case 1: Per hours number of concurrent users (a) and requests throughput (b).](image-url)
First, for each case study, we have determined the alternative optimal solution. In particular:

- the alternative initial solution is computed solving the optimization model in [30] with CPLEX 10.0,
- the performance model adopted is the one proposed in [32],
- the proportional assignment scheme is adopted as in [22], and
- the alternative initial solution is improved by the local search algorithm, switching servers in low power sleep/active states as in [32].

In this way, we can derive also the incoming request workload for our algorithm (right side of Figs. 3 and 4) which, we recall, is based on an open queuing model. This allows to obtain a fair comparison with [32] where closed models are considered (the users think time has been randomly generated uniformly between 30 and 50 s). Still to get a fair comparison, we have not considered in our algorithm the DVFS move but we account for the energy costs of servers according to the power model in [20], where the energy cost is independent of the server utilization but it depends only on the CPU frequency.

Moreover, the objective function adopted in [22] has been reduced to a linear utility function and extended to include the cost of using servers. However, our objective function differs from the one adopted in [22] since the utility value is weighted by the throughput of the request classes. To compare the two different approaches, we have determined the requests response time by simulating the two different solutions and we have evaluated the two objective functions accordingly (see Fig. 5).

The obtained results show that our algorithm performs much better during the peak than the alternative solution which incurs in high penalties. Conversely, under light load, the two solutions provide similar results. Fig. 6 reports the response times obtained by simulation. Note that our solution provides response times one order of magnitude better than the alternative one. The motivation behind this big difference is that our solution mainly assigns dedicated servers to requests classes which is much more efficient than the proportional assignment scheme adopted in the alternative solution. Consider, for example, two servers with the same capacity and two request classes. For the sake of simplicity, assume that the two classes have the same arrival rate $\lambda$ and the same service time $1/\mu$. In the
proportional assignment scheme, the rate of execution for each class on each server is $\lambda_{11} = \lambda_{21} = \lambda_{12} = \lambda_{22} = \lambda/2$, while $\phi_{11} = \phi_{21} = \phi_{12} = \phi_{22} = 0.5$; thus, the response time for each class (see (2)) is $R_1 = R_2 = 1/(0.5(\mu - \lambda))$. In the dedicated server scheme, the rate of execution for each class on each server is, e.g., $\lambda_{11} = \lambda_{12} = \lambda$ and $\lambda_{21} = \lambda_{22} = 0$, while $\phi_{11} = \phi_{12} = 1$ and $\phi_{21} = \phi_{22} = 0$. Therefore, the response time for each class is $R_1 = R_2 = 1/(\mu - \lambda)$, hence half of the one relative to the proportional scheme.

Figs. 7 and 8 show the number of servers in active state obtained in the two solutions. Our solution uses the resources more efficiently since, excluding the initial part of the day, it provides better response times while adopting a lower number of servers. Recall that for a fair comparison with [22], we are not considering the DVFS, then the energy cost reduction is proportional to the difference on the number of servers in active state. Overall, during the 24 hours, our solution gives net revenues higher than the alternative one which is never profitable with respect to both the two objective functions.

Computational results relative to case study 2 are reported in Figs. 8, 9, and 10. The improvement which can be obtained by our solution over the 24 hours is 45 percent with respect to the alternative objective function. The improvement is very significant also relatively to our objective function: during the 24 hours, our overall profit is 3.8 while the alternative solution incurs in -103.5 penalty. Also in this case, our solution always provides better response time while using a lower number of servers (see Fig. 8).

### 6.2 Test in a Real Prototype Environment

The effectiveness of our resource management policies on a real system has been evaluated performing experiments running the JSP implementation of the SPECweb2005 and the RUBBoS benchmarks. SPECweb2005 is the industry standard benchmark for the performance assessment of webservers. We have considered the e-commerce and banking workloads, which simulate the access to an online trading and to an online banking website implementing an HTTPS/HTTP mix and HTTPS only load, respectively. RUBBoS is an open source benchmark implementing an online bulletin board system with pure HTTP requests. SPECweb2005 includes four components: the load generators, the client coordinator, the webservice, and the back-end simulator. The SPECweb2005 load generators inject workload to the system.
According to a closed model, user sessions are started according to a given number of users who continuously send requests for dynamic webpages, wait for an average think time $Z = 10$ s, and then access another page or leave the system according to a predefined session profile. The client coordinator initializes all the other systems, monitors the test, and collects the results. The webservice is the component target of the performance assessment (Apache Tomcat 5.5.27 in our setup), while the back-end simulator emulates the database and application parts of the benchmark and is used to determine the dynamic content of the webpages. Overall, the benchmark allows assessing the performance of single tiers web systems, since the back-end simulator is a dummy component.

Similarly, RUBBoS includes three components: the client emulator, the webservice container (Apache Tomcat 5.5.57), and a DBMS (MySQL 5.0). As in SPECweb2005, the workload is generated according to a closed model and we set the think time $Z = 7$ s. RUBBoS allows emulating two tier systems, since the second tier implements a real database storing two years of stories and comments.

Our experimental setup includes three physical servers: the first server hosts the load generators, while the remaining two are the System Under Tests (SUTs) target of our analyses and host the web and DBMS server instances. The SUTs are two homogeneous servers based on Intel Nehalem dual socket quad-core CPUs with 16 GB of RAM which run Xen 3.4.1-rc10. Each core has five P-states, corresponding to frequencies of 2.4, 2.27, 2.13, 2.00, and 1.6 GHz. Since our system has similar characteristics to the one considered in [24], here we have adopted the same power model which is reported in Fig. 2c. Core P-states can be set independently and are updated by the xenpm tool.7

7. In our performance model, we have introduced a queue for each physical server in the service center. Modeling each core as a queue leads to more accurate results. However, our solution is general since, even if the technology does not allow scaling the frequency of cores independently, it is easy to extend problem (P1) to consider the cores available inside physical servers and to add new constraints which entail assigning the same frequency to the cores implemented in the same physical socket. Actually, physical servers contain several resources, e.g., CPUs, disks, network interface, etc. An enhancement of our performance model, where the resources within a server are modeled as a network of queues, is part of future work.

The VMs hosting the web and DBMS servers run Linux Ubuntu Server 8.10.

The Xen hypervisor is assigned to a dedicated core on the first socket of each server, while the VMs running the web and DBMS instances are assigned to the cores of the second socket by setting CPU affinity. The SPECweb2005 back-end simulators are assigned to a dedicated core on the second socket; they are always overprovisioned and their performance will be neglected. The Xen hypervisor runs the default credit scheduler which is a proportional fair share CPU scheduler [36]. Xen provides two parameters which determine resource settings: Weight and cap. A VM with a weight of 512 will get twice as much CPU as a VM with a weight of 256 on a contended host. Legal weights range from 1 to 65,535 and the default is 256. The cap optionally fixes the maximum amount of CPU that a VM will be able to consume, even if the host system has idle CPU cycles. In our experiment, we have not set caps and, for each VM, the weights have been set proportionally to VM’s $\phi$. Furthermore, 2 GB of RAM has been reserved to each VM. We have obtained an estimate of $\mu_{k,j}$ parameters of the applications under study by assigning VMs to dedicated cores and performing an extensive offline profiling along the lines in [21].

The validation in the real testbed is based on two case studies. In the first one (case 3), a two classes single tier system based on SPECweb2005 e-commerce and banking workloads has been considered. We set $a_1 = a_2 = 1$, $R_1 = 1.4$ s, and $R_2 = 0.7$ s (reported as line dot plot in Figs. 12 and 14). Fig. 11 (plot left) reports the number of users considered during the experiment. The number of users $N_1$ of the first class (e-commerce) varies abruptly stepwise every five minutes, while for the second class (banking), $N_2$ remains constant and equal to 150. The class throughput, in terms of number of webpages per seconds, obtained during the experiment is reported in the plot on the right of Fig. 11.4

In the first time interval, the two classes are supported by two VMs which share a single core of the first SUT server

8. Our optimization framework is based on an open performance model: we have estimated the overall incoming workload a priori as $A_k = N_k/Z$, since in the considered number of users range, response times were significantly lower than the user think time (we recall that, for the response time law, $N_k = (R_k + Z_k) \cdot \lambda_k$).
running at the maximum frequency on the first socket. After
5 minutes, we have considered a worst case scenario of
100 percent increase of class 1 workload, and we run the
long-term algorithm. Then, another VM hosting the class 1
webserver is instantiated and assigned to a dedicated core
of the first SUT server which runs also at the maximum
available frequency. The response times of the two classes
are reported in Fig. 12. In the following 15 minutes only
values are updated every 5 minutes running the
short-term algorithm (the values are omitted for space
limitations). In the last time interval, \( N_1 \) is further reduced,
the three VMs remain active but the two cores are set to the
minimum frequency.

Figs. 11 and 12 show that our resource allocation policies
are effective since the system remains in the profitably
performance region and it is able to react to abrupt
workload variations.

In the second case study (case 4), the test includes three
request classes and lasts 8 hours. Classes 1 and 2 run
SPECweb2005 e-commerce and banking, respectively, while
class 3 executes RUBBoS. We set \( a_1 = a_3 = 5 \), \( a_2 = 10 \), \( R_1 = 0.85 \) s, \( R_2 = 0.40 \) s, and \( R_3 = 1.20 \) s. The plots on Fig. 13
report the number of users during the experiment. The
workload corresponds to the peak hours (10.00-18.00) of
the traces of the University website considered also in the
previous section (left side of Fig. 3). Data collected from the
log on a hourly basis have been interpolated linearly and
oversampled adding also some noise to obtain workload
traces varying every 10 minutes as in [19], [20]. The short-
and long-term solutions are executed every 10 and
30 minutes, respectively. Initially, server 1 hosts on a first
core three VMs running the webserver for the three classes,
and on a second core, a VM running class 3 database. The
two cores are set to the maximum frequency P0. Server 2
hosts on two dedicated cores two additional instances of the
webserver for classes 1 and 2 which are set to P1.
The experiment can be characterized by four epochs: during the first epoch, which lasts 60 minutes, only the $\phi$ and $\lambda$ variables are updated. At the beginning of the second epoch, the long-term solution assigns the class 3 webserver to an additional dedicated core of server 1, while class 1 and 2 webserver VMs still share computing capacity on the initial core. Furthermore, an additional instance of the webserver is instantiated for class 2 which is assigned to a dedicated core on server 2. The second epoch lasts until minute 220 and updates only $\phi$ and $\lambda$ variables. At that time instant, the workload decreases and the cores of the two servers are set to P3 and P4, respectively. At minute 260, the fourth epoch starts raising the frequency again of the two servers to P1 and P2, respectively. Then, this configuration is held until the end of the test, updating only VMs’ capacity allocation for class 1 and 2 webservers sharing server 1 core, and the load balancing for the five running webserver instances of classes 1 and 2. Fig. 14 shows the average requests response times estimated by the performance model and measured in the prototyping environment. Also in this case, the values are almost always below the thresholds and the system is kept in the profitably region.

### 6.3 Energy Savings

To evaluate the energy savings which can be achieved by our solution, we have considered a large service center with 400 servers and 100 requests on four tiers. We have investigated two different workload case studies, referred to as cases 5 and 6, obtained from case 1 and case 2, respectively, where from each request class, the incoming workload of 10 classes has been obtained by adding random noise in each control time interval. The comparison has been made with a service center always running at full capacity. We have considered only energy cost, while SLA revenues have been neglected. The computational results, reported in Fig. 15, show that significant energy savings can be obtained in case 5 under light load conditions. Over the 24 hours, the consumption of energy can be reduced by 33 percent. In case 6 (see Fig. 16), the savings are more uniform during the day and the overall energy cost reduction during the 24 hours is almost 17 percent.

### 6.4 Sensitivity to Workload Prediction

In the computational experiments, we have considered the overall requests incoming workload as given: $\Lambda_k$ results from the prediction of the exogenous arrival rates for the next long (short) term control time horizon and the admission control mechanism (see Section 3). To account
for the effect of workload unpredictability, as in [19], we have evaluated the performance of our long-term resource allocation policies comparing the results obtained by our algorithm with the results obtained by an oracle that has a perfect knowledge of the future. Since homogeneous systems are characterized by lower variability, Table 2 reports sensitivity results for heterogeneous systems only.

The number of request classes |K|, the number of servers |I|, and the capacity ratio CR have been varied and the values selected are reported in the first two columns. \( \Lambda_k \) values for all k have been varied simultaneously by \( \pm 5 \), \( \pm 10 \), and \( \pm 15 \)%. Results show that the gap between the objective function value determined by the oracle and the one obtained by our long-term algorithm under uncertainty ranges between 20% and 30% (10% in average). The objective function percentage variation is independent of the system capacity ratio CR, showing that our algorithm is able to determine optimal tradeoff between SLA revenues and energy costs both for light and heavy workloads.

### 7 Conclusions

We have proposed resource allocation policies for multiclass virtualized environments maximizing the profits associated with multiple-class SLAs. The cost model consists of a class of utility functions which include revenues and penalties incurred depending on the achieved level of performance and the energy costs associated with the use of physical servers. The overall optimization problem involves the set of servers to be turned in active state, servers operating frequencies, the allocation of VMs to servers, the load balancing, and capacity allocation of VMs. In contrast to previous literature proposals, those variables have been considered in a unifying framework. The corresponding optimization problem is NP-hard and a heuristic procedure has been proposed. The solution is provided for multiple-time scales and it is effective even for large size problem instances. Systems up to 400 servers and 100 request classes can be managed very efficiently. The effectiveness of our approach has been assessed by performing simulation and experiments in a real prototype environment. Synthetic as well as realistic workloads and a number of different scenarios of interest have been considered. Furthermore, a comparison with top-performing state-of-the-art techniques shows that our solutions outperform alternative methods providing net revenues up to 45 percent higher. Moreover, solutions are robust to server and workload variations. The solution is particular effective for managing CPU bounded applications under high workload conditions. Future work will analyze the interrelationships which characterize the short- and long-term problems combining the proposed utility-based approach with pure control theoretic techniques.

### Acknowledgments

The authors would like to thank Dr. Ramya Raghavendra, Dr. Parthasarathy Ranganathan, Dr. Vanish Talwar, Dr. Zhikui Wang, and Dr. Xiaoyun Zhu for providing the server power models used in their experimental analysis. Thanks are also expressed to Dr. Folco Bombardieri for development and experimental activities. The work of Danilo Ardagna and Barbara Panicucci has been partially supported by the GAME-IT research project funded by Politecnico di Milano. The work of Danilo Ardagna has been supported also by the European Commission in the context of the Q-ImPrESS research project (http://www.q-impress.eu) under the ICT priority of the Seventh Research Framework Programme.

### References


