Dynamic Resource Allocation in Computing Clouds through Distributed Multiple Criteria Decision Analysis

Yağız Onat Yazır[α], Chris Matthews[α], Roozbeh Farahbod[β]
Stephen Neville[γ], Adel Guitouni[β], Sudhakar Ganti[α] and Yvonne Coady[α]

[α]
Dept. of Computer Science
University of Victoria
Victoria, BC, Canada
{onat, cmatthew, sganti, ycoady}@cs.uvic.ca

[β]
C2 Decision Support System Section
Defense R&D Canada Valcartier
Quebec, QC, Canada
{firstname.lastname}@drdc-rddc.gc.ca
&
Faculty of Business
University of Victoria
Victoria, BC, Canada

[γ]
Dept. of Electrical and Computer Engineering
University of Victoria
Victoria, BC, Canada
sneville@ece.uvic.ca

***Abstract*—In computing clouds, it is desirable to avoid wasting resources as a result of under-utilization and to avoid lengthy response times as a result of over-utilization. In this technical report, we investigate a new approach for dynamic autonomous resource management in computing clouds. The main contribution of this work is two-fold. First, we adopt a distributed architecture where resource management is decomposed into independent tasks, each of which is performed by Autonomous Node Agents that are tightly coupled with the physical machines in a data center. Second, the Autonomous Node Agents carry out configurations in parallel through Multiple Criteria Decision Analysis using the PROMETHEE method. Simulation results show that the proposed approach is promising in terms of scalability, feasibility and flexibility.**

I. Introduction

Cloud computing is a popular trend in current computing which attempts to provide cheap and easy access to computational resources. Compared to previous paradigms, cloud computing focuses on treating computational resources as measurable and billable utilities. From the clients' point of view, cloud computing provides an abstraction of the underlying hardware architecture. This abstraction saves them the costs of design, setup and maintenance of a data center to host their Application Environments (AE). Whereas for cloud *providers*, the arrangement yields an opportunity to profit by hosting many AEs. This economy of scale provides benefits to both parties, but leaves the providers in a position where they must have an efficient and cost effective data center.

Infrastructure as a Service (IaaS) cloud computing focuses on providing a computing infrastructure that leverages system virtualization [1] to allow multiple Virtual Machines (VM) to be consolidated on one Physical Machine (PM) where VMs often represent components of AEs. VMs are loosely coupled with the PM they are running on; as a result, not only can a VM be started on any PM, but also, it can be migrated to other PMs in the data center. Migrations can either be accomplished by temporarily suspending the VM and transferring it, or by means of a live migration in which the VM is only stopped for a split second [2]. With the current technologies, migrations can be performed on the order of seconds to minutes depending on the size and activity of the VM to be migrated and the network bandwidth between the two.

The ability to migrate VMs makes it possible to dynamically adjust data center utilization and tune the resources allocated to AEs. Furthermore, these adjustments can be automated through formally defined strategies in order to continuously manage the resources in a data center with less human intervention. We referred to this task as the process of *dynamic and autonomous resource management* in a data center.

In former research, this process is generally carried out through a centralized architecture. The research focuses on the use of utility functions to declare the preferences of AEs over a range of resource levels in terms of utilities. The utility values are then communicated to a global arbiter that computes and performs the resource management on behalf of the entire data center [3].

In this paper, we propose a new approach to the same problem in the context of computing clouds. Our contribution is two-fold. Firstly, we adopt a distributed architecture where resource management is decomposed into independent tasks

and each task is performed by Autonomous Node Agents (NA) that are tightly coupled with the PMs in a data center. Secondly, NAs carry out configurations through Multiple Criteria Decision Analysis (MCDA) using the PROMETHEE method [4], [5].

The rest of this report is organized as follows. In Section II, we provide a general description of the problem. Section III outlines the former research in the field. Section IV provides the details of the overall design our approach. In Section V we provide details on the PROMETHEE method. In Section VI, we explain the experimental setup in detail. Section VII provides a detailed assessment of our approach. Finally, Section VIII summarizes our conclusions and outlines our future directions.

II. Problem Description

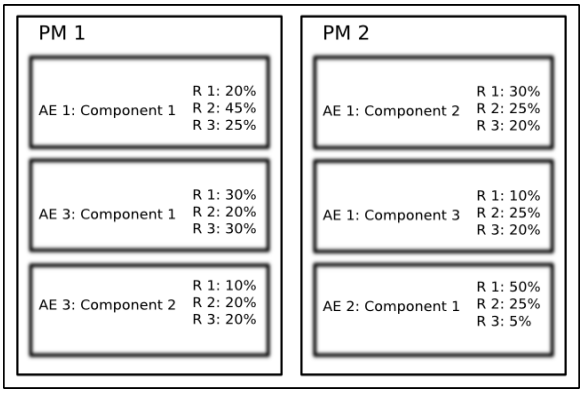
Resource consolidation in data centers can generally be defined as the *dynamic* and *autonomous* process of providing mappings between a set of application environments and a pool of shared computational resources.

The process is dynamic since the resource usages of the application environments are time variant which stems from two major factors. First of all, the amount of computational resources is variable throughout time due to the possibility of addition, removal and temporary unavailability of hardware/software in a data center (e.g. unexpected failures, scheduled maintenance, etc.). Secondly, neither the number of application environments nor their performance level and resource level requirements at an arbitrary instance can be statically determined. While variability in the number of application environments in a data center arise from registration and de-registration of application environments (e.g. arrival and departure), variability in requirements are due to the changes in performance level requirements (e.g. upgrading or downgrading) and fluctuations in resource level requirements (e.g. increasing or decreasing workload). An example of the time variance in a data center is illustrated in Figure 1 in terms of changing resource requirements and arriving/departing application environments. Resource consolidation is also an autonomous process in a sense that its goal is to minimize human intervention during mapping/re-mapping in order to provide a robust self-configuring infrastructure that is more responsive to changing conditions.

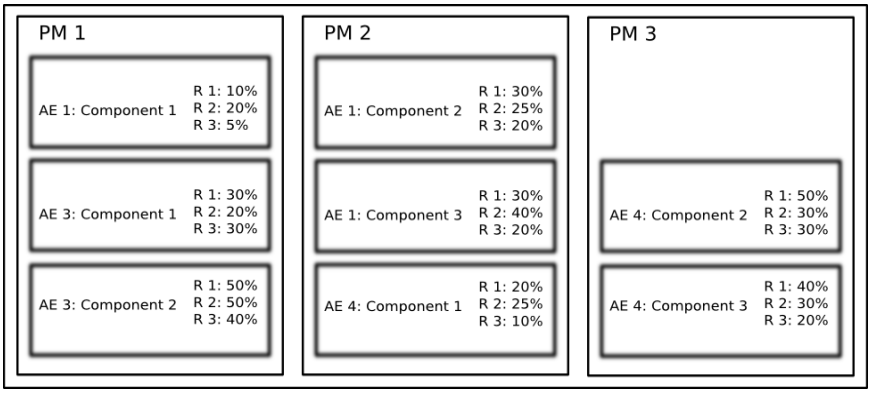
The primary actors that benefit from resource consolidation are *clients* and *providers*. Clients can simply be defined as the owners of application environments that execute within a data center. From the clients' point of view a data center is merely a pool of resources. One of the expectations of clients is to have a level of abstraction from the low-level operational details of the infrastructure (e.g. servers, switches, topology, software deployment, dynamic allocation of resources, etc.), through which they will be able to define the desired performance—generally referred to as *quality of service* or *service level* agreements—of their application environments using high-level performance metrics. Based on these high level definitions, clients assume that their application environments are assigned sufficient amounts of computational resources, so that, their performance level requirements are continuously met regardless of the changes in their workload. Moreover, it is also natural for clients to claim a certain degree of control over the quality of their application environments through modification of the high-level performance parameters for various reasons at any point in time (e.g. increasing/decreasing service quality).

On the other hand, providers are basically the owners of data centers and the corresponding infrastructure that facilitates provisioning of resources among multiple application environments. The central expectation of providers is to maximize profit which can be defined in terms of *revenue* and *expenses* as $P = R - E$. Revenue, R, is in general a function of the number of application environments hosted in the data center, and the continuity of success in meeting their performance level requirements. E is a function of expenses such as purchasing and maintenance of computational units, power consumption, cooling, deployment areas, cabling and docking equipment, taxes, etc. [6]. Generally, all of the listed expenses are in fact directly proportional with the number of computational units in a data center. This in turn implies that E can be modeled as a function of the number of nodes. Therefore, providers' central goal is to compute the least number of computational units that can successfully host the highest number of application environments without violating the service level agreements at a given instance in order to minimize *expenses* and maximize *revenue*—hence maximize the *profit*. This, in essence, can be viewed as a problem of continuously balancing the data center utilization, which has been one of the central focuses in the field since pointed out by Almeida et. al. as a criteria for both short-term and long-term resource planning in data centers [7].

In this context, a resource consolidator must facilitate a certain set of functionality. The first of these is a reasonable mapping between performance level and resource level requirements per application environment in order to find out the amount of resources that need to be allocated for each application environment to meet its desired performance. At this point, it is plausible to assume that these metrics are predefined by the provider as a finite set of all possible high-level units of performance. Then, let us define such a set as $P = \{p_1, p_2, ..., p_k\}$, where each p_i represents a unique metric of performance. In the literature, examples of p_i are often outlined as throughput, latency, response time, average number of jobs per minute, etc. [3], [8], [9], [10], [11]. The fundamental assumption is that a client should be capable of defining any application environment in terms of the elements of P. This also implies that there is a finite set, $D_P = \{\delta_{P_1}, \delta_{P_2}, ..., \delta_{P_{2^k-1}}\}$, of all of the possible high-level quality definitions that can be provided by a client where each δ_{P_i} denotes a non-empty subset of P. On the other hand, at the lower levels, each application environment is defined by a finite set of physical resources, $R = \{r_1, r_2, ..., r_l\}$, where each r_i represents unique a resource metric. Generally, these metrics are defined as processing power, memory usage, hard drive usage, bandwidth, etc [3], [8], [9], [10], [11]. Then in a



(a) View of data center at time t.



(b) View of data center at time $t + k$, where $k > 0$.

Fig. 1. A view of a data center in time in terms of (1) the changes in the resource usages of application environments due to fluctuating workload, (2) arrival and departure of application environments, and (3) the changes in the number of computational units in the data center.

fashion similar to the definition of D_P, we can define a finite set of all possible physical resource definitions that an application environment can have, $D_R = \{\delta_{R_1}, \delta_{R_2}, ..., \delta_{R_{2^l-1}}\}$, where each δ_{R_i} denotes a unique non-empty subset of R. Then the mapping from performance requirements to resource requirements can be defined as a function $F_I : D_Q \rightarrow D_P$. It is crucial to define such a mapping for each application environment in order to generate suitable performance and resource usage models, which has attracted a considerable amount of interest in the research field [9], [10], [12], [13] mainly focusing on the idea of defining each F_I as a utility function—where preference ranking of different resource usages were considered rather than a direct quantification of sufficient resources—as first proposed by Walsh et. al. [3].

Once the application environments are defined in terms of their resources requirements, the consolidator is responsible for making sure that sufficient amounts of resources are continuously made available for each application environment. This requires monitoring and logging of resource usages of each application environment, detecting violations of performance requirements that are agreed upon by the clients and providers, and configuring/re-configuring resource allocations when necessary. In the light of these, the main functionality of a resource consolidator in a data center can be defined in terms of an observe-detect-react cycle respectively, as illustrated in Figure 2.

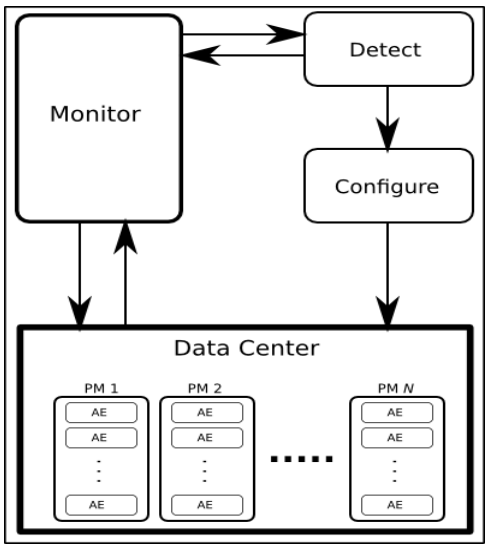


Fig. 2. Representation of a Resource Consolidator in terms of the phases of Monitor, Detect and Configure as an Observe-Detect-React cycle.

Monitoring is simply the phase where information regarding the resource utilization on each computational unit and the resource usage of each application environment in the data center is gathered. A generally employed method is based on periodically sampling data at the end of fixed time intervals. The raw information on resource usage that is collected by the monitoring mechanism is later processed in the detection phase. In the *detection* phase, the main concern is to capture any anomalies in terms of the violations of performance aggreements, and if there is any, to trigger the configuration phase. In the literature, the detection phase is devised in the two different manners of either treating it as a separate phase that is entered periodically [3], [9], [10], [14], or by merging it with monitoring to trigger configuration on an event-driven basis where events are often considered to be violations—or predicted violations based on demand and load forecasting—of performance agreements [15], [16], [17], [18], [15]. Finally, *configuration* is the phase where the consolidation reacts to the changes in the data center in terms of re-allocating resources for application environments that have undergone changes that have critical affects on their performances. As a result, application environments are re-assigned resources wherever available in the data center, which in turn might require movement of certain components of an application environment between computational units. The nature and effects of such movements are also have attracted a certain amount of interest in the field [19], [16], [17]. In the literature, this configuration phase is generally considered as a Multi-Dimensional Knapsack Problem or as a certain variant of it—

Vector Bin Packing Problem, which are NP-Hard. [3], [17].

III. Related Work

The former research in the field can be viewed under the two groups: (1) resource consolidation in multiple-server data centers, and (2) resource consolidation on a single-server shared centers.

Studies that fall under the first group focused on allocation of resources in a data center to a number of application environments. A two-layered architecture of a resource consolidation system for non-virtualized data centers was proposed in the work of Walsh et. al. [3], which is used as a common architecture in a majority of the later research. This architecture consists of a *local agent* assigned to each application environment, through which the required amount of resources is computed. The information that is generated by the local agents is then communicated to a *global arbiter* which computes a near-optimal reallocation of resources in the entire data center. This seminal work also integrated this two-layered approach with the usage of *utility functions* as a measure of desirability of different amount of resources from the application environments' point of view, where the utility values monotonically increase from undesirable amounts of resources to the desirable amounts of resources, between the values of 0 and 1. These utility values are then used to calculate a maximum global utility by the global arbiter under the condition of not exceeding the available resources in a data center. This, in essence, forms the optimization problem which is generally modeled as Knapsack Problem where a global near-optimal configuration is computed from the individual preferences of local agents. Studies outlined in Bennani and Menasce [10], Chess et. al. [8], Tesauro [12], Tesauro et. al. [9], [13], and Das et. al. [20] adopted this same approach which is then merged with forecasting methods based on different analytical models on non-virtualized data centers. While Bennani and Menasce [10] mainly focused on a queuing theoretic approach to the performance modeling problem, Tesauro [12], Tesauro et. al. [9], [13] considered a pure decompositional reinforcement learning approach and its hybridization with the queuing theoretic approach. In Chess et. al. and Das et. al. [8], [20], this same architecture and utility model is used to build a commercialized computing system, called *Unity*. Some of the common aspects of these works and their focus on resource consolidation are revealed as (1) a purely application environment centric view where data center utilization is not of major concern, (2) the main goal being the computation of an optimal configuration of resources in entire the data center, (3) and the absence of the cost of component movements in the formulation of the problem.

A number of research that falls under the same group have focused on virtualized data centers where the components of application environments are assumed to be encapsulated in virtual machines. Along with this approach, the data center utilization emerged as a major criterion during the course of dynamic allocations. The importance of data center utilization was first incorporated to a consolidator in the work of Almeida et. al. [7], and from that point on have become one of the central attributes of the research in the field. In this same work, utilization is shown to be a major factor in long-term resource planning in data centers alongside short-term planning achieved by resource consolidation. In other works, as outlined by Khanna et. al. [21] where data center utilization is not taken into account in contrast with other studies such as the ones outlined by Bobroff et. al. [14], Wood et. al. [19], Wang et. al. [16], Hermenier et. al. [17], and Van and Tran [18] where system architecture and the manner in which the allocations are performed are somewhat similar to the approach outlined in [3]. In a number of these, the cost of movements, known as migration, is taken into account as an attribute during the configuration of data center [19], [21], [16], [17], [18]. In Hermenier et. al. [17], the central tenet is constraint solving through the CHOCO library [22], where the general methodology is to find a set of possible configurations when re-allocation needs to be performed and electing the most suitable among them based on the least number of migrations necessary. This same methodology is also adopted in the follow-up work of Van and Tran [18].

The second group of research rather focused on distributing resources of a single computational unit among multiple application environments. In the work of Chandra et. al. [23], an analytical performance model is presented for generalized processor sharing on a single non-virtualilzed server combined with demand estimation methods to enhance the distribution of resources. Another similar approach is outlined in Mahabhashyam's study [24] where a single non-virtualized server distributes its resources among two application environments that are subject two different types of network transactions, streaming and elastic requests. Menasce and Bennani also examined the problem of autonomic allocation of CPU to multiple virtual machines is based on priority setting and time sharing using a beam search technique [25] to find the optimal distribution of resources [11].

The categorization of the studies on resource consolidation in multiple server data centers is outlined in Table I based on presence/absence of three main attributes: (1) consideration of resource utilization in a data center, (2) consideration of the cost of movements during reconfiguration, and (3) focusing on the goal of global optimization during reconfigurations. Note that, table also distinguishes between virtualized and non-virtualized views as well as between single-server and multiple-server views of a data center.

Although these methods are very efficient in relatively small data centers, we believe that they can potentially suffer from scalability, feasibility and flexibility issues.

The scalability issues may arise mainly due to centralized control and aim for near-optimal configurations. As the size of a data center increase the computation of a near-optimal configuration becomes more complex and time consuming. The time taken during such computations may result in stale configurations since the environment is time variant. The feasibility issues are also related to the aim for near-optimal configurations. Such configurations often require a total re-

TABLE I
CATEGORIZATION OF RELATED WORK ON RESOURCE CONSOLIDATION IN MULTIPLE SERVER DATA CENTERS

	Non-Virtualized	Virtualized
Resource Utilization	————	Almeida et. al., Wood et. al. Bobroff et. al., Hermenier et. al. Wang et. al., Van et. al.
Migrations Cost	————	Hermenier et. al. Van et. al. Khanna et. al.
Global Configuration	Walsh et. al., Tesauro et. al. Bennani et. al., Chess et. al. Tesauro, Das et. al.	Almeida et. al., Wood et. al. Bobroff et. al., Hermenier et. al. Wang et. al., Van et. al.

assignment of resources in the data center. This may result in a very high number of migrations rendering the application of the configuration impossible in a large data center. In addition, usage of utility functions may raise flexibility issues. In a large data center with many AEs, defining a common utility function to effectively represent all AEs' preferences over resources can be very complex. Moreover, the addition of new criteria requires a redefinition of the utility function to be used.

In this paper, we propose a new approach to the resource allocation problem through an architecture that distributes the responsibility of configuration among Node Agents (NA). Each NAs is a unit that is tightly coupled with a single PM in the data center, which configures its resources through MCDA only when imposed by the local conditions. The distribution of responsibility makes our approach inherently scalable. This limits the solution space to the local views NAs, which results in fast and up-to-date solutions regardless of the size of the data center. Since our approach does not aim for a global reconfiguration of the resources in the entire data center, the number of migrations per configuration is substantially less than the global solutions making our approach more feasible given current technology. Finally, NAs use the PROMETHEE method which gives us the flexibility particularly in terms of adding new criteria to the assessment of configurations along with the ability to easily tune the weights of criteria.

IV. SYSTEM ARCHITECTURE

We designed the system architecture based on the idea of avoiding the problems of scalability that may emerge as a result of determining and maintaining globally optimal configurations through facilitating a centralized arbiter. We strongly believe that as the data center expands both the computation of optimal configurations and centralizing these computations in a global arbiter might impose serious complexities. As a result of this view, we designed the system as a two-level architecture of (1) application agents that are closely coupled with application environments that declare up-to-date resource requirements, and (2) node agents that are coupled with the computational units in the data center that continuously distribute resources based on the data that application agents declared.

Each application environment in the system is assigned an application agent upon arrival to the data center. The main resposibility of the application agents is to continously provide the resource requirements that match the performance requirements of the application environments given certain initial performance requirements. Most importantly, the result of these mappings are subject to change as the workloads of application environments vary. Therefore, an application agent should closely monitor the workload of its corresponding application environment and update the resources necessary for it to meet the performance requirements outlined upon arival to the data center. We also assume, in parallel with the related work, that each application environment will have at least one component during the course of its lifetime, where each component performs a certain task within the objectives of a certain application environments. In our work, each of these components are assumed to be virtual machines to be deployed on computational units. The results of these mappings can be based both on the current workloads and the anticipated future workloads [9], [13], [20], on a per component basis. Note that, this agent is somewhat an equivalent of the local agents defined in the previous studies.

In the second layer of the architecture, however, instead of using a global arbiter to determine the configuration of resources, we assign this responsibility to node agents that are attahed to each computational unit in the data center. Each node agent has the responsibility of monitoring the changes to the resource requirements as declared by the application agents of the corrsponding virtual components that are currently being hosted on the computational unit that the node agent is attached to. Moreover, node agent perform the necessary configurations as the changes in resource requirements of the components impose. These changes in our view is of two natures: (1) Local re-distribution of the resources as long as the the computational can accommodate, and (2) Moving suitable components to other computational units when the current components can not be accommodated with the resources of

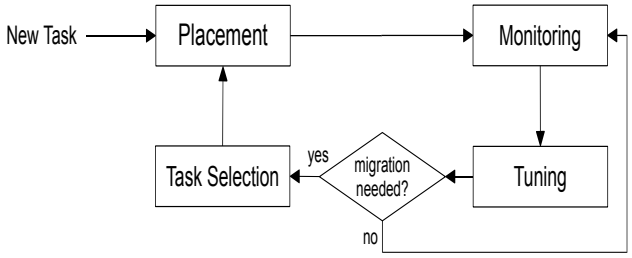


Fig. 3. Node Program

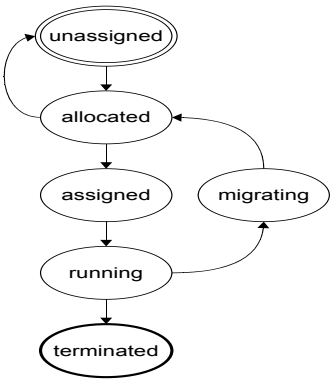


Fig. 4. Task Lifecycle

the hosting computational unit.

It is important to note here that the major focus of this paper is based on the behaviour of the node agents during the course of configurations, rather than the modelling of the performance and the mapping function of application environments, for we think that the behaviour of application agents should be investigated in a separate work. Accordingly, in the rest of this paper, we outline the abstract and mathematical model of the node agents.

In this context, we designed the system as a fully distributed network of autonomous node agents each capable of accommodating components—will be called running tasks in the rest of this paper—and, when necessary, delegating tasks to other nodes and handing over the management to the corresponding node agents. We assume that the network maintains a global awareness of the resource availability of nodes and their task assignments. In practice, this awareness can be achieved either by using already established protocols or by having nodes report to a centralized (or a hierarchy) of monitoring units.

The process of dynamically allocating tasks to nodes and maintaining resource distribution among tasks to meet their resource requirements, is modeled as a distributed process carried out by individual node agents in the system. Conceptually, every node—in parallel to running the tasks assigned to it—continuously performs a cycle of four activities (see Figure 3): *placement*, where a suitable node capable of running the given task is found and the task is assigned to that node; *monitoring* where the node monitors its tasks and resource requirements as declared by application agents; *tuning* where the node attempts to adjust its resource assignments locally in respond to changes in task resource requirements and determines if local accommodation is possible; *task selection* where, if local accommodation is not possible, a task is selected to be migrated to another node and the process loops back into placement.

We model every node as a DASM agent that continuously runs its main program. The following ASM program abstractly captures the behavior of the node agents.[1]. However, in parallel to the four main activities, every node also continuously responds to communication messages it receives from other nodes.

[1]Note that according to ASM semantics, the composition of the four activities in this program denotes a parallel execution of these activities

Node Program

NodeProgram ≡
  Placement
  MonitoringAndTuning
  if *migrationNeeded* **then** TaskSelection
  // the selected task will be migrated in the next cycle
  Communication

Since the system is fully distributed, new tasks can be assigned to any node in the system. Once entered into the network, a task goes through a lifecycle starting with being *unassigned* and ending with being *terminated* which is when its execution is completed (see Figure 4). For any given task at a given time, one node is responsible to move the task through its lifecycle. The current status of tasks are captured by the function *taskStatus* : TASK $\mapsto$ TASKSTATUS. Every node has a pool of tasks that it is responsible for, captured by the function *taskPool* : NODE $\mapsto$ SET(TASK). The set of tasks that are running or to be run on a node (assigned to the node for execution) is captured by the function *nodeAssignment* : NODE $\mapsto$ SET(TASK). Then, we have:

$$\forall n \in \text{NODE}\ \textit{nodeAssignment}(n) \subset \textit{taskPool}(n)$$

The rest of this section focuses on the main activities of the nodes.

A. Placement

The Placement activity consists of two tasks: *a)* finding suitable nodes and allocating resources for *unassigned* tasks, and *b)* assigning *allocated* tasks to their corresponding nodes. In the Placement activity, a node looks for the tasks that are either *unassigned* or *allocated*. For these tasks, the node is acting like a broker;

Placement

Placement ≡
  ResourceAllocation
  TaskAssignment

In ResourceAllocation, the node picks an unassigned task from its task pool and tries to find a suitable node with available resources that can perform the task. At this level, we capture the computation of finding such a node by the

abstract function *placementNode* : TASK $\mapsto$ NODE that given a task suggests node that can run the task. We have,

$$\begin{aligned} &\mathit{placementNode}(t) = n \in \text{NODE} \mid \\ &\qquad \mathit{hasRequiredResources}(n,t) \\ &\qquad \wedge\, \mathit{maxResourceUtilization}(n,t) \end{aligned}$$

placementNode first filters out the nodes in the data center that do not have the resources to host a certain task t. After a list of the nodes that have the necessary resources is provided, the problem is to choose the best node defined on two criteria, resource utilization and rate of change in resource usage. Resource utilization is the criteria that ensures that maximization of resource utilization is taken into account (lower the resources on a node, higher the utilization will be), whereas rate of change in resource usage reflects on the stability of available resource usage on a node (low rate of change points to less likeliness of further migrations, and thus less likeliness of performance drops on the tasks). In order to reflect on both the provider and clients these two criterion can be weighted respectively (taking into account both or one more than the other).

If such a node is found, the broker node sends a resource lock request to the target node to allocate its resources before actually delegating the task to the target node. It also changes the status of the task to *allocated*. The following ASM rule captures this behavior:

Resource Allocation

```
ResourceAllocation ≡
  choose t in taskpool(self) with taskStatus(t) = unassigned do
    let n = placementNode(t) in
      if n ≠ undef then
        RequestResourceLock(n, t)
        potentialNode(t) := n
        taskStatus(t) := allocated
        SetTimer(self, t)
      else
        Warning("Cannot find a suitable node.")
```

When a node receives a resource lock request for a given task, it checks whether it still has the required resources available. If so, it puts a lock on the given resources, sets a timer and sends an acknowledgement to the requesting node; otherwise, it sends a lock rejection. If it does not receive the task within a certain time window, it removes the lock and releases the resources.

Upon receiving a lock acknowledgement, the node changes the status of the task to *assigned* and sends the task to the target node. As part of the Communication activity, nodes continuously monitor incoming network messages. If a node receives a task-assignment message with a task that it has alread allocated resources for, it will release the lock, assigns the resources to the task, starts the task and adds it to the set of its assigned tasks.

Process Task Assignment

```
ProcessTaskAssignment(msg) ≡
  // assuming that msg is of type task-assignment
  let task = msgData(m) in
    add task to taskpool(self)
    if task ∈ allocatedTasks(self) then
        AssignResourcesToTask(task)
        ReleaseLock(task)
        add task to nodeAssignments(self)
        StartTask(task)// sets its status to running
        remove task from allocatedTasks(self)
    else
        taskStatus(task) := unassigned
```

Notice that at this level the node assumes the responsibility of the given task and if it doesn't have resources allocated (locked) for that task (which could be the case for new tasks), it simply sets the task status to *unassigned* and keeps it in its task pool so that it can be later processed by the Placement activity.

B. Monitoring And Tuning

Nodes continuously monitor their resource utilization and their task requirements. There are three main activities under monitoring and tuning: *a)* for all running tasks, a node monitors changes in its resource requirements and adjusts resource allocations if needed; if such adjustment is not possible, a task replacement may be triggered; *b)* a node also monitors its resource utilization and adjust its resource allocation to keep the utilization within a reasonable boundary; *c)* finally, any resource lock that is timed out (for which a task is not received) needs to be relased. The following ASM rule abstractly captures these three activities:

Monitoring and Tuning

```
MonitoringAndTuning ≡
  forall t in nodeAssignments(self) with taskStatus(t) = running do
    if taskRequirementChanged(t) then
      AccommodateRequirementChange(t)
  if resourceBoundReached(t) then
    Tuning
  ReleaseTimedoutResourceLocks
```

V. CONFIGURATION MODEL BASED ON A FORMAL DECISION MAKING METHOD

In this section we focus on the problem of configuration in terms of moving components of application environments between computational units. As outlined in the previous section, this is an action that is performed only when a node agent determines that it is no longer possible to accommodate all of the components that it is currently hosting. Then the problem was defined in an abstract manner in two basic operations: (1) Deciding on one or more suitable task(s) remove, and (2) Deciding where in the data center to move them to.

The main goal of the first decision—task selection—can be given as the necessity to bring the resource usage on the computational unit within the limits of available resources. However, the choice is not only dependent on this single criteria. It also needs to take into account minimizing the

cost of removal and minimizing the probability of further movements of that task from its new host. In this sense, the problem of choice becomes of multiple criteria nature. Then the task selection can be modelled as a multi-criteria decision making problem, where the decision maker is the node agent that detects the need for removal of component(s), and the criteria being maximizing the resource utilization on the computing unit, minimizing the time and the negative effects of the removal, and minimizing the probability of re-removal of the component from its new host. The alternatives can be outlined as the components that can bring the resource usage under acceptable limits (e.g. limits of the host). Furthermore, the decision method to be facilitated should provide means to remove multiple components when necessary without extra the computational burden of re-calculating a best choice.

The main goal of the second decision—node selection—can be given as determining a node that can accommodate the compoenent chosen in the first decision. This, in the first place depends on the amount of resources that a node can provide for the component. Thus the candidate set is defined as the nodes in the data center that can accommodate the chosen components. The resources should be the minimal—but still within limits of resources—on the chosen candidate in order to ensure that the overall utilization will not be low. However, as in the first case it is necessary to define a more comprehensive set of factors that affect this process. The most visible criteria other than the resources in that sense are probability of accommodating the chosen component without the need for further removals from the node, and minimizing the effects of the movement on the currently hosted components.

Then these two steps can be considered as multi criteria decision problem as given in:

$$max\{g_1(a), g_2(a), ..., g_j(a), ..., g_k(a) | a \in A\} \tag{1}$$

where A is a finite set of possible candidates $\{a_1, a_2, ...a_j, ...a_n\}$ and $\{g_1(.), g_2(.), ..., g_j(.), ..., g_k(.)\}$ a set of evaluation criteria. In our case some of the criteria will be maximized while some of them will be minimized based on the step of decision. Then, the decision maker expects to identify a candidate that optimises all the criteria.

In order to deal with this problem, we use the PROMETHEE preference modelling [4], a specific family of outranking methods. PROMETHEE methods were designed to treat multi-criteria problems of type 1 and their associated evaluation table. The information requested to run PROMETHEE consists of: (1) Information between criteria, and (2) Information within each criteria.

Information between criteria relates to the weights of each criteria in the decision problem as a measure of their relative importance. These weights, $\{w_j, j = 1, 2, ..., k\}$, are non-negative numbers, independent from measurement units of the criteria. The higher the weight, the more important the criterion during course of decision making. In general, there is no objection to using normalized weights: $\Sigma_{j=1}^{k} w_j = 1$. Assigning weights to the criteria in our design is the responsibility of the providers in a sense that these weights represent the providers' preferences as to how the resources should be consolidated (e.g. taking maximizing resource utilization, minimizing agreement vilolation, or a balance between them as the ultimate goal).

Information within criteria relates to the comparison of based on how good an alternative with respect to the others based on a percriteria basis. In contrast with Utility Functions, PROMETHEE does not allocate an intrinsic utility to each alternative, neither globally, nor on each criterion. Instead, the preference structure is based on pairwise comparisons. In this case the deviation between the evaluations of alternatives on a particular criterion is considered. These preferences can be considered as real numbers varying between 0 and 1. Then the node agent—the decision maker in general—facilitates a function for each criterion:

$$P_j(a, b) = F_j[d_j(a, b)] \forall a, b \in A, \tag{2}$$

where:

$$d_j(a, b) = g_j(a) - g_j(b) \tag{3}$$

and for which:

$$0 \leq P_j(a, b) \leq 1. \tag{4}$$

In case of a criterion to be maximized, this function provides the preference of a over b for observed deviations between their evaluations on criterion $g_j(.)$. For criteria to be minimized, the preference function should be reversed or alternatively given by:

$$P_j(a, b) = F_j[-d_j(a, b)] \forall a, b \in A, \tag{5}$$

The pair $\{g_j(.), P_j(a, b)\}$ is called the *generalized criterion* associated to criterion $g_j(.)$. Sucha generalized criterion has to be defined for each criterion. In order to facilitate this identification, six types of particular preference functions have been proposed: (1) Usual Criterion, (2) U-Shape Criterion, (3) V-Shape Criterion, (4) Level Criterion, (5) V-Shape with Indifference Criterion, and (6) Gaussian Criterion.

In order to further detail on how the alternatives are evaluated with respect to each other let us define the *Aggregated Indices* and the *Outranking Flows* as defined in PROMETHEE.

For aggregated indices, let $a, b \in A$, and let:

$$\begin{cases} \pi(a, b) = \Sigma_{j=1}^{k} P_j(a, b) w_j, \\ \pi(b, a) = \Sigma_{j=1}^{k} P_j(b, a) w_j, \end{cases} \tag{6}$$

$\pi(a, b)$ is expressing with what degree a is preferred to b over all the criteria and $\pi b, a$ with what degree b is preferred to a. In most cases there are criteria for which a is better than b, and criteria for which b is better than a, consequently $\pi(a, b)$ and $\pi(b, a)$ are usually positive. The following properties hold for all $(a, b) \in A$:

$$\begin{cases} \pi(a,a) = 0, \\ 0 \leq \pi(a,b) \leq 1, \\ 0 \leq \pi(b,a) \leq 1, 0 \leq \pi(a,b) + \pi(b,a) \leq 1, \end{cases} \tag{7}$$

And it is clear that,

$$\begin{cases} \pi(a,b) \ 0 \text{ implies a weak global preference of } a \text{ over } b, \\ \pi(a,b) \ 1 \text{ implies a strong global preference of } a \text{ over } b, \end{cases} \tag{8}$$

The outranking flows can be defined as follows. Each alternative a in A face $(n-1)$ other alternatives in A. The, let us define the outranking flows as positive and negative outranking flows, where positive outranking flow of a:

$$\phi^+(a) = \frac{1}{n-1} \Sigma_{x \in A}(\pi(a,x)), \tag{9}$$

and negative outranking flow of a:

$$\phi^-(a) = \frac{1}{n-1} \Sigma_{x \in A}(\pi(x,a)), \tag{10}$$

The positive outranking flow represents how an alternative is outranking all the others, meaning its *power* or *outranking character*. Thus, the higher $\phi^+(a)$, the better the alternative. Whereas, the negative outranking flow represents how an alternative is outranked by all others, meaning its *weakness* or *outranked character*. Thus, the lower $\phi^+(a)$, the better the alternative.

In PROMETHEE I: Partial Ranking, (P^I, I^I, R^I) is obtained from the positive and negative outranking flows where both flows do not usually induce the same rankings. In essence, PROMETHEE I is their intersection:

$$\begin{cases} aP^Ib \text{ iff } \begin{cases} \phi^+(a) > \phi^+(b) \text{ and } \phi^-(a) < \phi^-(b), \text{ or} \\ \phi^+(a) = \phi^+(b) \text{ and } \phi^-(a) < \phi^-(b), \text{ or} \\ \phi^+(a) > \phi^+(b) \text{ and } \phi^-(a) = \phi^-(b); \end{cases} \\ aI^Ib \text{ iff } \phi^+(a) = \phi^+(b) \text{ and } \phi^-(a) = \phi^-(b); \\ aR^Ib \text{ iff } \begin{cases} \phi^+(a) > \phi^+(b) \text{ and } \phi^-(a) > \phi^-(b), \text{ or} \\ \phi^+(a) < \phi^+(b) \text{ and } \phi^-(a) < \phi^-(b); \end{cases} \end{cases} \tag{11}$$

where P^I, I^I, and R^I stand for preference, indifference and incomparability respectively. When aP^Ib, a higher power of a is associated to a lower weakness of a with respect to b. The information of both outranking flows is consistent and may therefore be considered as sure. When aI^Ib, both negative and positive flows are equal. When aR^Ib, a higher power of one alternative is associated to a lower weakness of the other. This often happens when a is good on a set of criteria on which b is weak, and reversely, b is good on some criteria on which a is weak. In this last case, the information provided by both flows is not consistent. It is then reasonable to consider these two alternatives as incomparable. PROMETHEE I is prudent; it will not decide which action is best in such cases. It is up to the decision maker to take responsibility of choice. However, in our case, since one of the goals is to automate the process of consolidation in order to minimize human intervention, we focus on PROMETHEE II: Complete Ranking to resolve incomparabilities with the trade-off of information loss during the decision process.

PROMETHEE II consists of the (P^I, I^I) complete ranking. In order to provide such a complete ranking, the net outranking flow is considered:

$$\phi(a) = \phi^+(a) - \phi^-(a). \tag{12}$$

It is the balance between the positive and negative outranking flows. Thus, the higher the net flow, the better the alternative, so that:

$$\begin{cases} aP^Ib \text{ iff } \phi(a) > \phi(b), \\ aI^Ib \text{ iff } \phi(a) = \phi(b). \end{cases} \tag{13}$$

Also, the following properties hold:

$$\begin{cases} -1 \leq \phi(a) \leq 1, \\ \Sigma_{x \in A} \phi(a) = 0. \end{cases} \tag{14}$$

When PROMETHEE II is considered, all the alternatives are comparable. However, the resulting information can be more disputable because more information gets lost by considering the difference. In essence, since the net flow provides a complete ranking, it may be compared with a utility function, however, with the advantage of working with clear and simple preference information (e.g. weights and preference functions), and relies on comparative statements rather than absoulte ones. One other advantage of PROMETHEE II that directly applies to our approach is the simplicity in adding more ciriteria due to further considerations in the design and implementation of the consolidator, rather than building a new utility functions as new criteria are revealed. This can greatly simplify updates to our approach.

In the following two sections, we investigate the suitability of our methodology by providing the experimental setup and the results of using PROMETHEE II for resource configuration in a fairly large data center.

VI. Experimental Setup

We have built a simulation environment using the C programming language in order to test our approach and compare it to certain other strategies. The environment simulates the changes and configuration actions in a datacenter in a stepwise manner, where between each step there is a fixed time interval. We have chosen this view in order to be able to compare each strategy to be tested on the simulation environment at any instance of the simulation with identical conditions. Accordingly, every virtual component declare new resource usages at each step and the strategy that is being used to configure the data center reacts with respect to the new conditions. We define the resource dimensions as CPU usage, memory usage and bandwidth usage.

Layer one consists of modules that represent the virtual components and physical machines. Each virtual component in the simulation environment represents a task that is independent of the others, and each of those tasks are responsible to update their resource requirements at each step. Virtual components in our simulation environment are considered in two groups: batch and online processes. Virtual components of batch type update their resource requirements as random noise signals based on certain distributions (e.g. uniform, poisson, exponential, normal, pareto, etc.). Each virtual component update their resource requirements independently on each resource dimension where each dimension may have different distributions. Note that an application environment can be represented by a number of virtual components, and in this case an application environment that represents a batch process is assumed to be represented by one or more of batch-type virtual components. The online type virtual components work in *on* and *off* periods. The *on* periods represent the times when there are requests to be serviced, and the *off* periods represent idle intervals. During their *on* periods online virtual components generate resource requirements in a way that is identical to the batch tasks, while during the *off* periods they do not pose any recource requirements. The distribution of the length of *on* and *periods* are modeled to reflect the self-similar behaviour seen in online environments, that is, if one of the periods' duration is exponentially (or poisson) distributed the other has a heavy-tailed distribution (e.g. pareto). In the simulation environment each virtual component has a certain life time determined when they arrive. Life time of a virtual component is defined either in terms of a certain number of steps or as *unlimited* which means that it will reside in the data center forever. The physical machines represent bins that can acommodate one or more virtual components. In the simulation environment it is assumed that a virtual component cannot span multiple physical machines. Each physical machine keeps a record of the virtual components it hosts and the amount of resources in use by its virtual components. In addition, physical machines have an upper bound for resource usages to ensure that they do not have more virtual machines than they can acommodate and response times are kept at reasonable levels. Finally, they are capable of migrating their virtual components to other physical machines when necessary.

Layer two consists of a single module, the global monitor. The global monitor keeps a record of each virtual component and each physical machine in the system. While the virtual components are recorded only as allocated or not allocated without the details of their resource usages, physical machines in the system are recorded with respect to their available resources in each dimension. The second responsibility of this module is to reply to the queries about the states of the physical machines in the data center. Some of the typical queries that are replied to by the global monitor are *the first physical machine with the necessary resources*, *non-empty physical machine(s) that have a certain level of available resources on each dimension*, *empty physical machines*, etc.

Layer three consists of the strategy modules that represent different configuration methods that are used to consolidate resources. Every strategy module has to assign arriving virtual components to physical machines and re-configure the data center in terms of facilitating necessary movements to keep each virtual component below their upper bounds. The assignments and re-configuration is performed at each step during the simulation. We have implemented four strategy modules with two of them represent a centralized control that aims for a global near-optimal configuration, while the other two represent distributed control with local configuration. By local configuration we mean that the nodes are not concerned with the global consolidation but are focused on holding their resource usages below their resource upper bounds.

The two methods we implemented to represent the centralized control and global configuration strategy are the well-known bin packing First Fit (FF) and First Fit Decreasing (FFD) methods. FF performs configuration at each step by iterating through the list of virtual components that are present in the data center—either allocated or not allocated—and placing each of them on the first physical machine that can accommodate. FFD performs the exact same operation at each step with the difference of first sorting the list of virtual components in a decreasing order, that is, the virtual components that have the greater resource requirements get placed first. It is important to note here that in our implementation the sorting on multiple dimensions is done through assigning the same weight to all three of the dimensions. The methods that represent the distributed control and local configuration strategy consists of a very simple method and our approach. Briefly the simple method works as follows. When a physical machine is above its resource upper bound it picks the first virtual component in its list and places it at the first physical machine available reported by the global monitor. The initial placement is done similarly with the absence of the virtual component picking step. If a physical node is below the resource lower bound, then it attemopts to remove all of its virtual components and turn itself off. This virtual component to move first and the physical machine choices are made the same way as the upper bound case.

In our approach also, the physical machines are responsible for local configurations. We adapted PROMETHEE II as a decision making procedure for determining the configurations. The configuration is performed only in two cases: (1) Resource upper bound is reached, and (2) Resource lower bound is reached. When the upper bound is reached a physical agent tries to move the most suitable virtual component(s) in order to bring the resource usage below the upper bound again. This is performed using PROMETHEE II as follows. First, the physical machine agent filters a list of virtual components that can bring the resource usages below the upper bound, if this list is non-empty, then it performs the pairwise comparisons in order to sort the alternatives based on their net outranking flows. During these pairwise comparisons, the criteria are resource usages in each dimension and resource usage variabilities in each dimension. The variabilities are determined based on a

certain set of most recent data on resource usages. The weight of these criteria is left to the providers preference based on their goals. If the list is empty, then the physical machine agent uses all of the virtual components for the pairwise comparisons and the sorting of the net outranking flows of each alternative. Once the most suitable virtual component is picked, the physical machine agent contacts the global monitor in order to retreive a set of nodes that can acommodate the chosen virtual component. Using the same criteria, the most suitable one is picked from the set of alternative physical machines. Finally, the machine with resources above the upper bound moves the chosen virtual component to the chosen physical machine. In the case where there is no physical machine that can acommodate the virtual component in the data center, the cycle is repeated for the second best virtual component. If none of the virtual components can be replaced within the data center, a new node is created for the most suitable virtual component. In the case where a physical machines resource usages are below a lower bound, the physical machine agent—as in the simple method—tries to remove its virtual components and turn itself off. During this process, the virtual machines are not picked based on any criteria, however for the choice of destination the pairwise comparisons are again used. It is important to note here that, for preference functions we have used the Usual Criterion as outlined in Mareschal [4].

In the next section we provide detailed explanations of the tests that we have conducted and the results on the robustness, flexibility, efficiency and practicality of our model and its comparison to the other three methods mentioned in this section.

VII. Results and Validation

We have created test scenarios based on the number of virtual components to be hosted on a data center and the necessary size of the data center in terms of the physical machines to be used. In that sense, we focus on three distinct data centers with a certain number of virtual components: (1) Extra Small scenario represents a data center with 600 physical machines and 5000 virtual components to be hosted, (2) Small scenario represnts a data center with 1300 physical machines and 10000 virtual components, and (3) Medium scenario represents a data center with 2500 physical machines and 20000 virtual components. Please note that in, our tests cases the number of virtual components that can be hosted on physical machine is 10, which is a choice that we have made with respect to the maximum resource requirements that a virtual component can have. In each of the scenarios, our simulator uses all four methods explained in Section VI, and runs steps five times more than the number of virtual components. That is, if we have x virtual components to be hosted in a scenario, the simulation runs for $5 \times x$ steps. Figures 5, 6, 7 represent number of physical machines used and the number of migrations for each method in the extra small, small and medium secenarios respectively. It is also important to underline here that (this may go one chapter back), at any step each algorithm deals under the exact same conditions

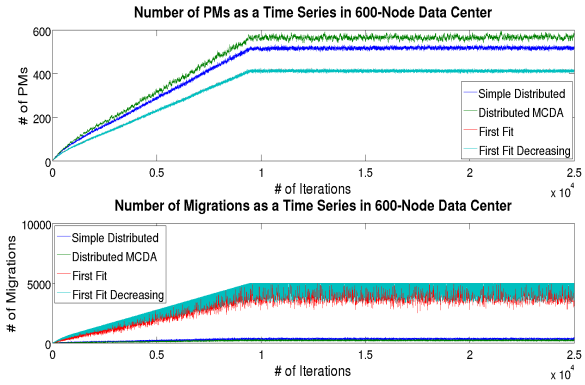


Fig. 5. Top figure represents the number of physical machines per algorithm that are in use at each step. The bottom figure represents the number of migrations per algorithm that are performed to configure the data center.

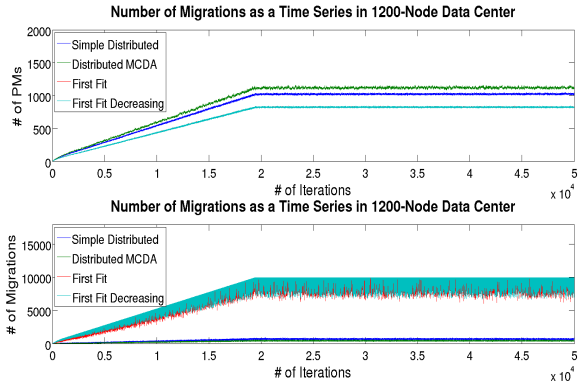


Fig. 6. Top figure represents the number of physical machines per algorithm that are in use at each step. The bottom figure represents the number of migrations per algorithm that are performed to configure the data center.

(number of virtual components, resource requirements per virtual components). Also, note that each scenario starts with 0 virtual components at iteration 0 and iteration by iteration reaches the specified final number of virtual components, and the physical machines are not untilized upto 100%, instead we set the upper bounds per each method to 60% for feasible response times. When a virtual component with finite lifetime departs from the data center, it is replaced by a fresh virtual component arrival. We have chosen such an implementation to be able to evaluate the methods under a state that can be called *steady in terms of the number of virtual components*.

It can clearly be seen in these figures that the centralized control methods (FF and FFD) need a number of physical machines substantially less than the distributed (simple method and our approach) approaches. This means that the configuration in the centralized cases result in higher utilization of the data center. However, it is also cleary seen in the migration measures that the centralized methods are performing an unreasonable number of migrations to reach that utilization level. In the distributed cases, however, the number of migrations that are performed for configuration is much lower with a tradeoff on the number of physical

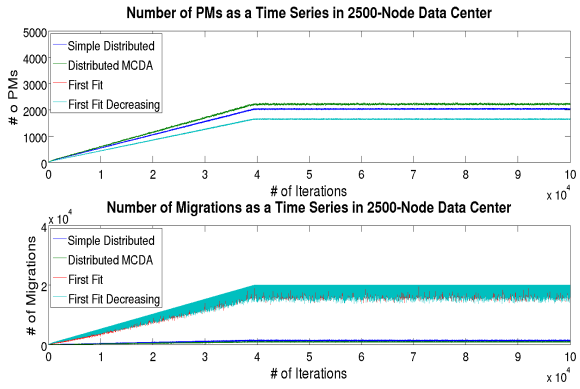


Fig. 7. Top figure represents the number of physical machines per algorithm that are in use at each step. The bottom figure represents the number of migrations per algorithm that are performed to configure the data center.

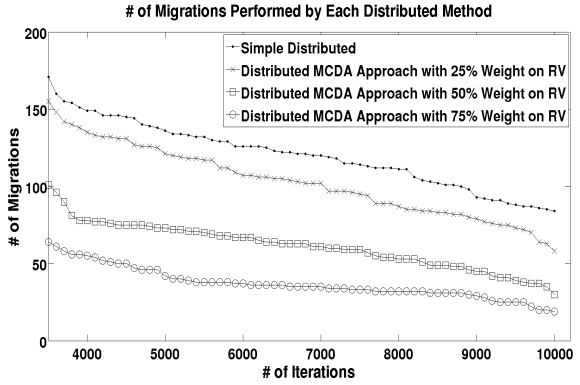


Fig. 8. The top line represents the number of migrations performed in the simple method in sorted format after steady state. The lines below that prepresent cases where our method uses more and more emphasis on resource variabilities at each dimension.

machines. When we take look at the distributed methods, it is seen that the simple method uses less physical machines than our approach, and our approach is performing less migrations in all of the scenarios. Again, this difference stems from the same tradeoff of using slightly more machines to make the data center more manageable in terms of the migrations. In the larger data centers the number of migrations that are performed by the simple method is substanially higher than the migrations that our approach performs. Please note that in all of these test cases, our method puts equal importance on the criteria.

However, the main advantage of our method lies in its ability to facilitate a flexibility in terms of adjusting the tradeoff between having a reasonable number of migrations performed at each step and yet having a slight increase in the number opf physical machines. Figure 8 show that as we increase the weight of the resource variability criteria our approach performs lower number of migrations the effects of which on the number of physical machines can be viewed in Figure 9. Basically, as the weight on resource variablity criteria is increased the number of physical machines that are used in the test case slightly increases.

One important advantage of the distributed methods over

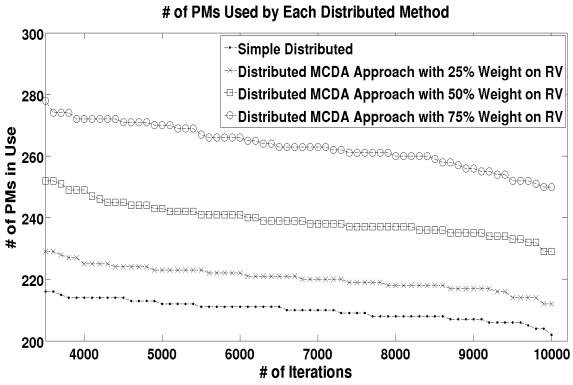


Fig. 9. The bottom line represents the number of PMs used in the simple method in sorted format after steady state. The lines above that prepresent cases where our method uses more and more emphasis on resource variabilities at each dimension.

the centralized ones can be investigated in the overall CPU, Memory and Bandwidth utilizations in the data center on per machine basis in Figures 10, 12, 11 respectively. For FF and FFD methods, CPU utilization as given in Figure 10show that the utilization is mostly around the upper bound as a plateau. This means that the CPU is utilization is very high in the data center because a low number physical machines are used. On the other hand, in the distributed methods, more machines are used to host the same number of virtual components, however, with a better distribution of utilization onver the data center. In essence, this means that the response times in the distributed methods will be lower, thus the likeliness of service level agreements violations decrease.

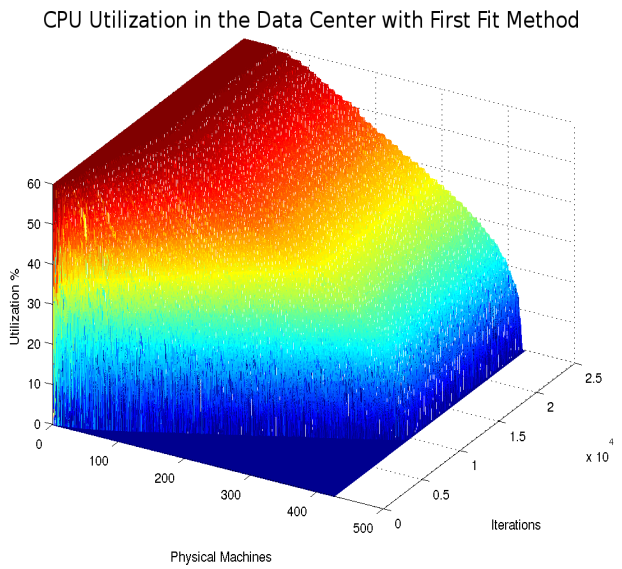
A similar distribution can be viewed for bandwidth utilization in Figure 11.

As per memory utilizations, the difference between the centalized cases adn distributed cases is more clear. Since the memory usages vary less then the CPU and bandwidth usages, the high utilization plateau in the FF and FFD methods are dominant. However, in the distributed cases, the utilization is distributed smoothly over the entire data center due to the usage of more machines.

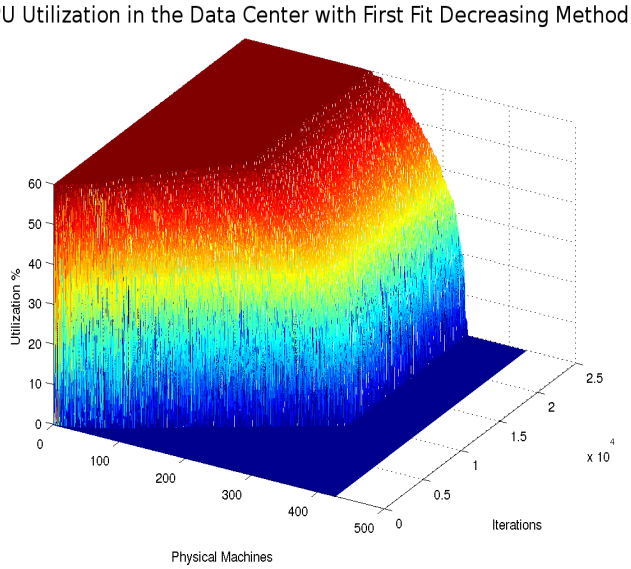
VIII. Conclusion and Future Work

In this paper, we introduced a new approach for dynamic autonomous resource management in computing clouds. Our approach consists of a distributed architecture of NAs that perform resource configurations using MCDA with the PROMETHEE method. The simulation results show that this approach is promising particularly with respect to scalability, feasibility and flexibility.

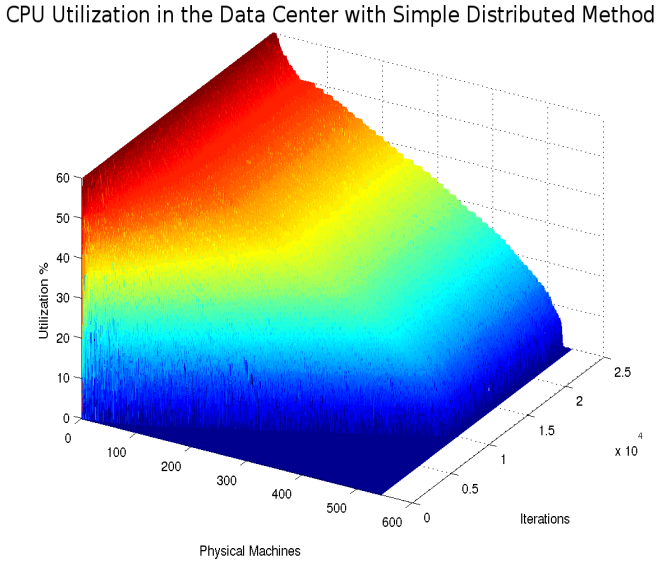
Scalability is achieved through a distributed approach that reduces the computational complexity of computing new configurations. Simulation results show that our approach is potentially more feasible in large data centers compared to centralized approaches. In essence, this feasibility is due to the significantly lower number of migrations that are performed in order to apply new configurations. Simulation results show that



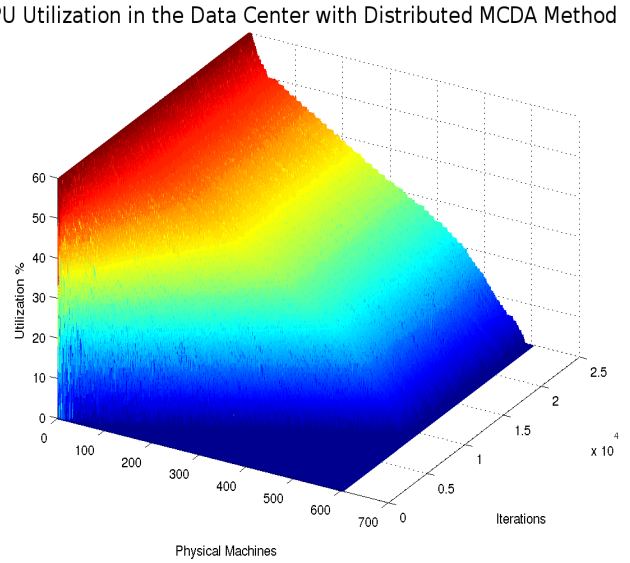
(a) View of overall CPU utilizations of the data center per physical machine when First Fit method is used.



(b) View of overall CPU utilizations of the data center per physical machine when First Fit Decreasing method is used.



(c) View of overall CPU utilizations of the data center per physical machine when the simple distributed method is used.



(d) View of overall CPU utilizations of the data center per physical machine when our distributed method is used.

Fig. 10. CPU utilization in the data center on a per machine basis.

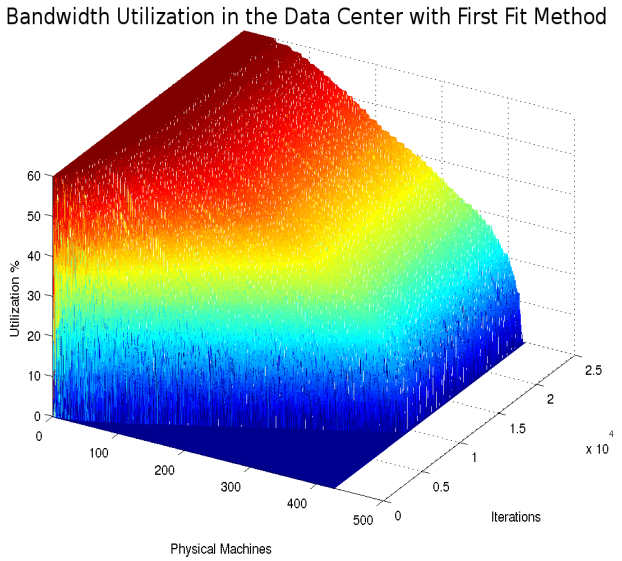
our method should theoretically trigger less SLA violations by smoothly distributing the overall resource utilization over slightly more PMs in the data center. The flexibility of our approach comes from its ability to easily change the weights of criteria and adding/removing criteria in order to change configuration goals.

In the next stages of this work, our goal is to include new criteria—such as VM activity—to reflect on the overhead of migrations more precisely. We are going to explore further refinements to our use of the PROMETHEE method by incorporating generalized criteria other than the Usual Criterion. In addition, we plan to compare the use of PROMETHEE to other MCDA methods. Finally, we are working on the design and implementation of new modules that will enhance the simulation environment with respect to the measurement of SLA violations.

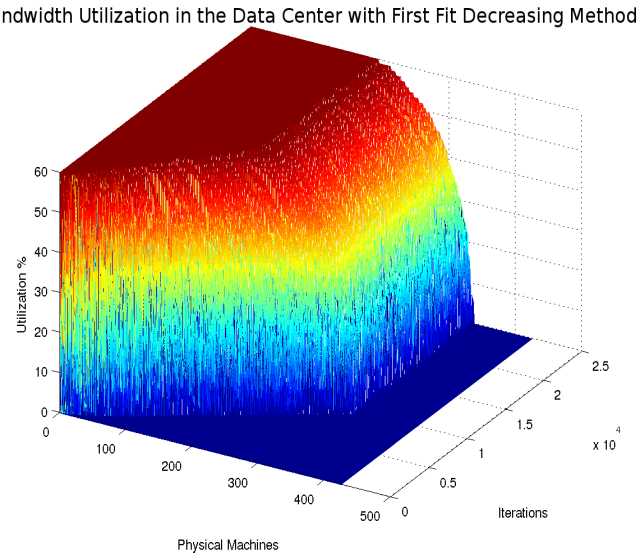
REFERENCES

[1] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *ACM Communications*, vol. 17, no. 7, pp. 412–421, 1974.

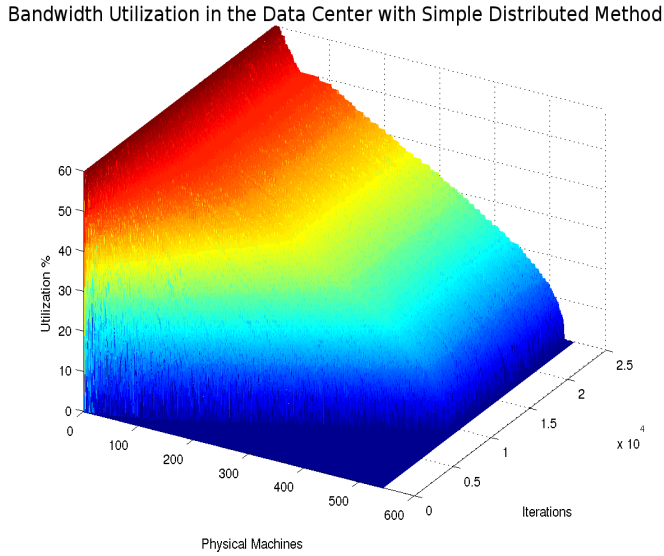
[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems*



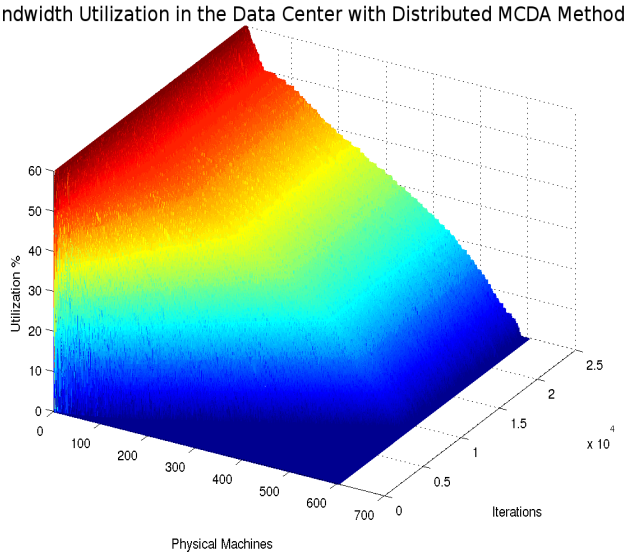
(a) View of overall Memory utilizations of the data center per physical machine when First Fit method is used.



(b) View of overall Memory utilizations of the data center per physical machine when First Fit Decreasing method is used.



(c) View of overall Memory utilizations of the data center per physical machine when the simple distributed method is used.



(d) View of overall Memory utilizations of the data center per physical machine when our distributed method is used.

Fig. 11. Bandwidth utilization in the data center on a per machine basis.

Design & Implementation. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[3] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility Functions in Autonomic Systems," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*. IEEE Computer Society, 2004, pp. 70–77.

[4] B. Mareschal, "Aide a la Decision Multicritere: Developpements Recents des Methodes PROMETHEE," *Cahiers du Centre d'Etudes en Recherche Operationelle*, pp. 175–241, 1987.

[5] J.-P. Brans, J. Figueira, S. Greco, and M. Ehrgott, *Multiple Criteria Decision Analysis:State of the Art Surveys*. Springer New York.

[6] J. G. Koomey, C. Belady, M. Patterson, A. Santos, and K.-D. Lange, "Assessing Trends over Time in Performance, Costs, and Energy Use for Servers," Intel and Microsoft, Tech. Rep., 2009.

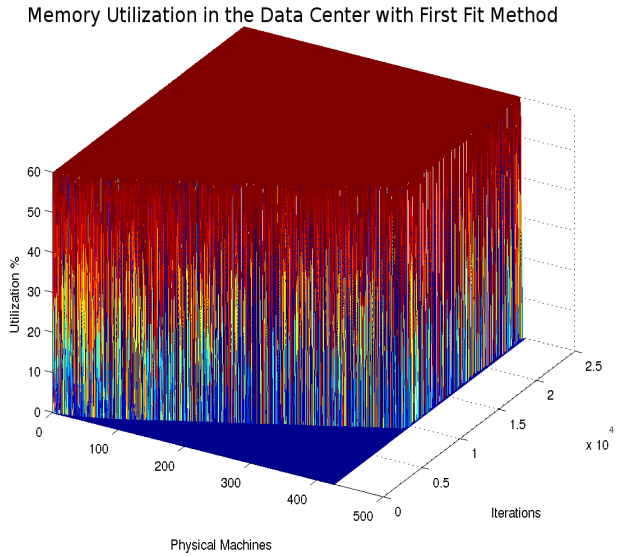
[7] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian, "Resource Management in the Autonomic Service-Oriented Architecture," in *ICAC '06: IEEE International Conference on Autonomic Computing*, 2006, pp. 84–92.

[8] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: Experiences with a Prototype Autonomic Computing System," in *2004. Proceedings. International Conference on Autonomic Computing*, 2004, pp. 140–147.

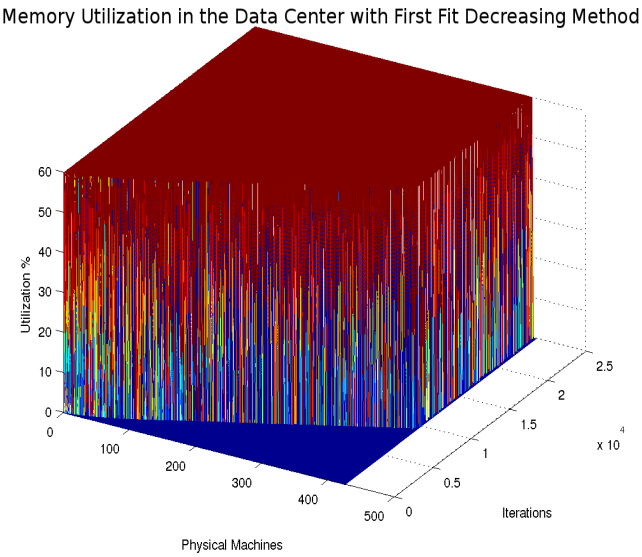
[9] G. Tesauro, R. Das, W. Walsh, and J. Kephart, "Utility-Function-Driven Resource Allocation in Autonomic Systems," in *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, 2005, pp. 342–343.

[10] M. N. Bennani and D. A. Menasce, "Resource Allocation for Autonomic Data Centers using Analytic Performance Models," in *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, 2005, pp. 229–240.

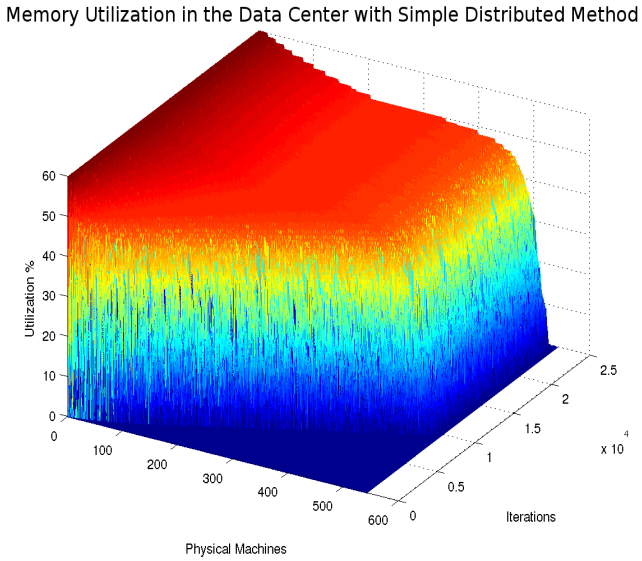
[11] D. A. Menasce and M. N. Bennani, "Autonomic Virtualized Envi-



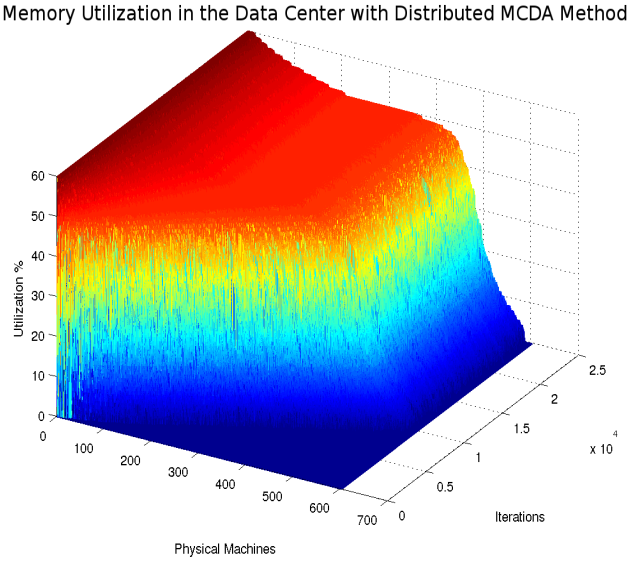
(a) View of overall Memory utilizations of the data center per physical machine when First Fit method is used.



(b) View of overall Memory utilizations of the data center per physical machine when First Fit Decreasing method is used.



(c) View of overall Memory utilizations of the data center per physical machine when the simple distributed method is used.



(d) View of overall Memory utilizations of the data center per physical machine when our distributed method is used.

Fig. 12. Memory utilization in the data center on a per machine basis.

ronments," in *ICAS: Proceedings of the International Conference on Autonomic and Autonomous Systems*. IEEE Computer Society, 2006, pp. 28–37.

[12] G. Tesauro, "Online Resource Allocation Using Decompositional Reinforcement Learning," in *AAAI'05: Proceedings of the 20th National Conference on Artificial Intelligence*. AAAI Press, 2005, pp. 886–891.

[13] G. Tesauro, N. Jong, R. Das, and M. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*. IEEE Computer Society, 2006, pp. 65–73.

[14] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic Placement of Virtual Machines for Managing SLA Violations," in *IM '07: 10th IFIP/IEEE International Symposium on Integrated Network Management*, 2007, pp. 119–128.

[15] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase, "Virtual Machine Hosting for Networked Clusters: Building the Foundations for Autonomic Orchestration," in *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, 2006, pp. 7–7.

[16] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Wang, "Appliance-Based Autonomic Provisioning Framework for Virtualized Outsourcing Data Center," in *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, 2007, pp. 29–29.

[17] F. Hermenier, X. Lorca, J. M. Menaud, G. Muller, and J. Lawall, "Entropy: A Consolidation Manager for Clusters," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2009, pp. 41–50.

[18] H. N. Van and F. D. Tran, "Autonomic Virtual Resource Management for Service Hosting Platforms," in *ICES '09: Proceedings of the International Conference on Software Enginnering Workshop on Software Engineering Challenges of Cloud Computing*. IEEE Computer Society,

2009, pp. 1–8.

[19] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-Box and Gray-Box Strategies for Virtual Machine Migration," in *4th USENIX Symposium on Networked Systems Design and Implementation*, 2007, pp. 229–242.

[20] R. Das, J. Kephart, I. Whalley, and P. Vytas, "Towards Commercialization of Utility-based Resource Allocation," in *ICAC '06: IEEE International Conference on Autonomic Computing*, 2006, pp. 287–290.

[21] G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application Performance Management in Virtualized Server Environments," in *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, 2006, pp. 373–381.

[22] N. Jussien, G. Rochart, and X. Lorca, "The CHOCO ConstraintProgramming Solver," in *CPAIOR'08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08)*, 2008.

[23] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 300–301, 2003.

[24] S. R. Mahabhashyam, "Dynamic Resource Allocation of Shared Data Centers Supporting Multiclass Requests," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*. IEEE Computer Society, 2004, pp. 222–229.

[25] V. J. Rayward-Smith, I. H. Osman, and C. R. Reeves, *Modern Heuristic Search Methods*. John Wiley and Sons, 1996.