Revenue Driven Resource Allocation for Virtualized Data Centers

Sajib Kundu[◇] Raju Rangaswami[†] Ming Zhao[†] Ajay Gulati[§] Kaushik Dutta[‡]
[◇]Riverbed Technology [†]Florida International University [§]ZeroStack, Inc. [‡]National University of Singapore

***Abstract*—The increasing VM density in cloud hosting services makes careful management of physical resources such as CPU, memory, and I/O bandwidth within individual virtualized servers a priority. To maximize cost-efficiency, resource management needs to be coupled with the revenue generating mechanisms of cloud hosting: the service level agreements (SLAs) of hosted client applications. In this paper, we develop a server resource management framework that reduces data center resource management complexity substantially. Our solution implements revenue-driven dynamic resource allocation which continuously steers the resource distribution across hosted VMs within a server such as to maximize the SLA-generated revenue from the server. Our experimental evaluation for a VMware ESX hypervisor highlights the importance of both resource isolation and resource sharing across VMs. The empirical data shows a 7%-54% increase in total revenue generated for a mix of 10-25 VMs hosting either similar or diverse workloads when compared to using the currently available resource distribution mechanisms in ESX.**

I. Introduction

In cloud hosting data centers, tens to hundreds of virtual machines with diverse characteristics can be consolidated in one physical server. Higher degrees of consolidation are attractive because it optimizes the utilization of server resources. On the downside, managing high VM density well is non-trivial. Over-sized VMs present problems in terms of capital and operational expenditure while under-provisioned VMs result in violation of the service level agreement (SLA) and customer dissatisfaction.

In Infrastructure-as-a-Service (IaaS) cloud data centers, clients host their applications inside VMs and manage the applications themselves. The allocation and management of individual VM resources, on the other hand, are performed by the cloud service providers. Clients pay rent to the service providers for use of the server resources.

Cloud hosting services today use simple *service level agreements* (SLA) whereby the clients are charged a flat fee based on the resource capacity they are renting or buying [1]. However, this is not ideal for either clients or data center service providers. For the customer, there is no easy way to determine an appropriate capacity since resource needs can depend on dynamic workload requirements. Consequently, they either pay more for unnecessary resources or risk performance violations. Data center service providers, on the other hand, use over-provisioning to avoid the heavy penalties associated with performance violations. *Performance-based* charging offers an alternative — clients pay rent for receiving specific levels of performance for their applications running within the hosted VMs [17]. The service providers can adopt resource provisioning schemes whereby applications are penalized/rewarded in terms of resource allocations as per the SLA-based revenue lost/generated. Performance-based charging also boosts clients' confidence in the cloud service since they pay only for performance required and experienced. Figure 1 presents a deployment scenario for performance-based charging wherein a set of co-located VMs generate a certain amount of revenue per unit time at a given performance level.

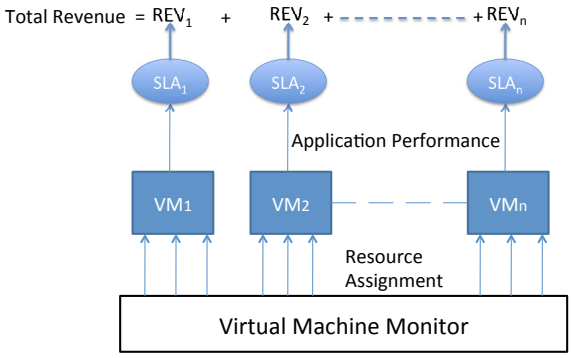


Fig. 1. System Model for Revenue Driven Resource Allocation.

In this paper, we propose a revenue-driven dynamic resource allocation solution which continuously steers the resource distribution across hosted VMs such as to maximize the collective revenue generated by the VMs at any given instant. This task is non-trivial because application performance depends on multiple resource types in complex ways and the per-application SLA curves can be quite diverse. Our previous work on *VM performance modeling* builds robust application performance models and lays the foundation for this work [16]. Since these performance models rely on the accuracy of observed input parameters such as a VM's storage I/O latency, abrupt and arbitrary reallotments of resources are not possible since doing so affects performance prediction accuracy. We take an incremental approach to dynamic resource allocation which relies on multiple incremental resource reallocation steps to attain the state of maximum revenue.

We implemented our solution in the VMware ESX hypervisor and evaluated it against both static and dynamic VM-level resource allocation mechanisms offered by ESX and the VMware vCenter Server management system. Our experimental evaluation with servers hosting 10-25 virtual machines highlights the importance of both resource isolation as well as resource sharing, although these principles are fundamentally at conflict with each other. By employing strict isolation across 10 functionally similar VMs in a VMware ESX server, our dynamic revenue maximization techniques converge to revenue levels that are 10-22% higher than configurations that distribute resources equally across VMs and a 7% increase over the existing work-conserving, share-based resource distribution mechanisms available in the vCenter Server management system [5]. Across a set of 25 functionally dissimilar VMs, our solution, when combined with the existing

share-based resource distribution mechanisms available in ESX hypervisors, registered 54% improvement in generated revenue relative to a solution that uses equal allocations. We make the following contributions in this paper:

1. We formalize the revenue-driven resource allocation problem for virtualized servers and prove it to be NP-hard.

2. We propose a practical heuristic solution which continuously, but incrementally, steers the resource distribution across hosted VMs within a server such as to maximize the SLA-generated revenue from the VMs.

3. We evaluate our approach for the VMware ESX hypervisor using the vCenter Server management system, demonstrating substantial increases in total revenue generation for both similar and diverse workloads and SLAs when compared against using the resource distribution mechanisms available in VMware ESX hypervisors.

II. Background

Our approach to dynamic VM resource management builds upon our previous work on *VM performance modeling* [16]. This previous work identified resource parameters for partitionable resources (e.g. CPU, memory) and for non-partitionable resources (e.g. storage, network) that can serve as control knobs for administrators to distribute physical resources. It also developed machine learning based models to robustly characterize the non-linear impact of resource allocation on an application VMs performance in the presence of contention for resources by other VMs. In this section, we briefly review this previous work and specifically discuss how the proposed models therein were used in our work.

A. Identifying Resource Parameters

The robustness of dynamic resource allocation depends on the successful identification of control knobs for key system resources and on the use of accurate application performance models. Our previous work on building robust VM performance models [16] used the following resource parameters that we use in this study as well. We use VMware ESX hypervisor's *CPU Limit in MHz* or *CPU Share* [11] for tuning CPU allocation to any VM. For memory, we use *Memory Limit in MB* or *Memory Share* for setting the memory allocation for any VM. Finally, based on the observation that storage is not easily partitionable, we use *VM I/O Latency*, an observed metric, as an input parameter for the performance models.

B. Building and Using Application Performance Models

Following the process described in our previous work [16], we built artificial neural network based performance models for each of the applications used in this study. These models, when used in production, predict the application performance given a set of resource assignments. While the performance models address how changes in resource availability impact application performance, they do not inform the extent of change necessary to achieve an optimization goal.

As we shall elaborate later, the distribution of multiple types of resources across VMs in a shared cluster with the objective of SLA-guided revenue maximization is an NP hard problem. The optimization framework that we propose in this paper uses performance models to maximize data-center revenue. Our incremental resource allocation solution also accounts for and addresses how changes to resource allocation affect other system variables such as I/O contention. More specifically, the pre-built application performance models are used to determine the increase or decrease in application performance for a given workload when a certain resource's availability is altered. An increase or decrease in application performance alters the corresponding SLA-generated revenue from the application which in turn informs how beneficial specific resource reallocation decisions would be.

III. Dynamic Resource Allocation

We model the problem of dynamic resource allocation in virtualized systems to explicitly take into account the influence of both resource allocation and resource competition. The model maps resource allocations of individual application VMs to revenues generated in US dollars as dictated by their respective SLA functions. Performance models are used for performance prediction within this optimization framework. We establish that the multi-resource reallocation problem for application VMs is at least NP-hard and that exact solutions are infeasible in practice.

It has been previously established that the performance of individual virtualized applications are determined by the resource assignments and current competition levels posed by other applications sharing the host [15], [17]. To determine the range of revenues generatable by a virtualized application at a future instant, the application performance under possible future resource assignments must be determined. The application performance models that we discussed in the previous section can help meet this need. However, a complete redistribution of resources is not possible when using these models since such redistribution would introduce large variance to the observed VM I/O latency and thus compromise the performance prediction accuracy of these models. In our solution, administrator-defined parameters, k and δ, serve to limit the magnitude of resource distribution by bounding the maximum allocation change that can be made for each resource type within a single resource reallocation operation. An application's performance obtained after model-based prediction is mapped to the application-generated revenue using the application-specific SLA curve. We assume that SLA curves can be complex, non-linear descriptions of revenue dependent on the application performance metric and that they are not necessarily restricted to simple priority assignments across applications.

A. Problem Formulation

Table I lists the parameters employed in the resource allocation problem formulation. The SLA-based optimal resource allocation problem can be formally specified as follows:

"Given $\mathbf{n}$ *application VMs (denoted as set I),* $\mathbf{m}$ *allocatable resource dimensions (denoted as set J), current resource allotments* $\mathbf{R}^{alloc}$*, performance models* $\mathbf{PM}$*, and SLA-based revenue function* $\mathbf{S(R)}$*, determine a set of new resource assignments* $\mathbf{R}^{opt}$ *for the VMs which will result in maximizing the total revenue REV generated across all VMs for certain*

Parameter	Description
n	Number of application VMs
m	Number of resource types
k	Maximum number of times the resource allocation of any VM can be changed for any resource dimension in a single resource reallocation operation
δ	A vector of length m denoting the units of changes in m resource dimensions.
I	Set of application VMs
J	Set of resource types
R^{alloc}	Current resource allocation vector of dimension m×n
R^{total}	A vector of length m for total available resources
$R_{i,j}$	Resource allocation for VM $i \in I$ of resource type $j \in J$
R^{opt}	Optimal resource allocation of vector m×n after the redistribution
S(R)	A vector of n SLA functions mapping the application performance to revenue in USD
REV	Revenue vector of length n
T	Time interval of running reallocation algorithm.
PM	A vector of length n, each member is a separate performance model for one VM App $i \in I$
$\Psi(R_{i,j}, \ \forall j \in J)$	Revenue for application $i \in I$, where amount of resources allocated to application i is $R_{i,j}, \forall j \in J$.

TABLE I. DESCRIPTION OF SYMBOLS USED IN RESOURCE ALLOCATION PROBLEM FORMULATION.

time interval **T***, given that any change to resource assignment* $\mathbf{R}_{i,j}$**,** $i \in I$**,** $j \in J$ *is bounded by* **-k**δ *to* **+k**δ*."*

At each interval T, the problem may be formalized as:

$$Maximize \quad REV = \sum_{i \in I} \Psi(R_{i,j}^{opt}, \ \forall j \in J) \tag{1}$$

subject to :

$$\textstyle\sum_{i \in I} R_{i,j}^{opt} \leq R_j^{total} \quad \forall j \in J \tag{2}$$

$$R_{i,j}^{opt} - R_{i,j}^{alloc} \leq \pm k\delta \quad \forall i \in I, j \in J \tag{3}$$

Equation 1 maximizes the revenue across all resources. Equation 2 restricts the total resource allocation for each resources across all application VMs to be less than the total available resources. Equation 3 restricts the resource allocation change of each resource type for each application VM to a maximum of $k\delta$.

The revenue derived by a data center from a particular application VM depends on the SLA and on the performance of the application, which in turn depends on the resources allocated to the application VM. However, all these dependencies are non-linear. Finally, $R_{i,j}$, R^{alloc}, R^{total} and R^{opt} are assumed to be integer values.

Theorem 1. *The resource allocation problem is at least NP-hard.*

Proof: Let us assume that the function Ψ is a linear summation function as follows $\Psi(R_{i,j}^{opt}, \ \forall j \in J) = \sum_{j \in J} A_j R_{i,j}^{opt}$, $\forall i \in I$, where A_js are constants. Let us also assume, $\delta = \infty$. Then the resource allocation problem reduces as follows:

$$Maximize \quad \sum_{i \in I} \sum_{j \in J} A_j R_{i,j}^{opt} \tag{4}$$

subject to :

$$\textstyle\sum_{i \in I} R_{i,j}^{opt} \leq R_j^{total} \quad \forall j \in J \tag{5}$$

The above problem is the **INTEGER KNAPSACK** problem, an established NP-Complete problem [13]. Since the **INTEGER KNAPSACK** problem can be reduced to a specific reduced instance of the resource allocation problem in polynomial time, we can conclude that this reduced subset problem of the resource allocation problem is NP-Complete. Consequently, with the additional constraints of Equation 3, the resource allocation problem is at least NP-hard. ■

B. How Expensive is Exhaustive Search?

Exhaustive search techniques may be applied to the resource allocation problem to find the most optimal solution. For many NP-hard problems, the small input size allows trivial, exact solutions in practice via exhaustive search; we examine if this is true for the problem under consideration. Per our problem formulation, each resource reallocation can assume *2k* different values, *k* positions for increments, and *k* positions for decrements. Comparing *2k* different possible changes in revenue values for each of the *n* VMs and for each of the *m* resource types to find R^{opt} will incur an asymptotic time-complexity of $O((2k)^{mn})$ which is infeasible even for small *n* and *k*. If we assume that $m = 4$, $n = 10$, and $k = 5$; the time taken to run brute-force search will take $O(10^{40})$ time units. Alternate, efficient heuristic solutions are thus needed for realistic deployment.

C. Other Heuristic Solutions

Both the resource allocation problem and the class of classic knapsack optimization problems have a common objective of determining a set of items to include in a sack of finite weight with a goal of maximizing the total value of the sacked items and with the constraint that the sum of all weights should be less than or equal to the capacity of the sack. Specifically, the total available capacity of any type of resource informs the capacity of the sack and the resource assignments of individual items provide the weights of the items selected to place in the sack and whereby the revenue from each VM is the value of each item. Maximizing total knapsack value maps to maximizing total revenue for the resource allocation problem. Given this similarity, we can consider applying heuristic solutions from the class of knapsack problems to the resource allocation problem.

However, despite its similarity to knapsack, the revenue maximization problem has several distinguishing characteristics. First, it is multi-dimensional with *m* resource types, which substantially increases the size of the solution search space. Second, because SLA-based revenue functions can be nonlinear, solutions to linear knapsack problems cannot be used as-is to solve our problem. Third, and the most important distinguishing feature, is that for the sake of system stability, any solution to the resource allocation problem must employ incremental resource adjustments, instead of reallocating resources from scratch, and such resource change is constrained according to Equation 3. This additional constraint makes resource allocation substantially more challenging precluding existing solutions to multi-dimensional, nonlinear knapsack problems [7]. Furthermore, the reduction of an instance of the dynamic resource allocation problem to a complex variant of knapsack is possible only by eliminating Equation 3. With the addition of Equation 3, the problem becomes unsolvable using existing approximation algorithms for the known knapsack family of problems.

IV. A Heuristic Solution

In this section, we present a dynamic resource allocation algorithm for revenue maximization with an acceptable time complexity. Given a current set of resource assignments for a pool of application VMs, the algorithm attempts to find a new set of allocations for each resource to maximize the total revenue generated at current application demand. In our current implementation, the algorithm runs periodically to make resource changes in an incremental fashion, with the goal of converging to a state of higher revenue at any given time. Periodic execution helps in multiple ways: it allows settling time for the resource changes that were made at the end of the previous execution and also allows capturing stable workload behavior [14]. Algorithm 1 lists the steps formally while Table II describes the parameters used. We quantify the effect of incremental changes using $\delta P^g_{i,j}$ and $\delta P^l_{i,j}$ which denote the gain or loss in revenue as the assignment of resource type *j* for application VM *i* is increased or decreased respectively by an amount δ.

Let us assume that the current resource allocation of resource type *j* for VM *i* is *r* which provides a revenue of *p* dollars. The application performance model predicts application output for an allocation of $r \pm \delta$ which is subsequently mapped by S_i to find the corresponding revenue p_δ. The difference between p_δ and *p* indicates the gain or loss for a δ increment or decrement, defined as $\delta P^g_{i,j}$ or $\delta P^l_{i,j}$. We assume that $\delta P^g_{i,j} \geq 0$, i.e., increasing resource allocation to a VM always results in no change or an increase in performance and consequently no change or an increase in revenue for the VM. On the other hand, $\delta P^l_{i,j} \leq 0$, i.e., taking away resources can not cause an increase in revenue.

The algorithm uses an iterative, greedy approach to revenue maximization. In each execution, it transfers resources from the VM that offers the least reduction in revenue due to a reduction in those resources to the most revenue-generating VM. In doing so, it also chooses the resource type for which the relative gain is maximized. Since in each execution, the algorithm makes incremental changes that increase overall revenue, subsequent periodic executions of the algorithm move the overall resource allocation state towards one that maximizes revenue for the system.

Algorithm 1 $MaxRevenue$: Revenue Maximization Algorithm

```
1: while (1) do
2:   MaxNetProfit = 0
3:   IC_{i,j} = 0 /* ∀i, j */
4:   DC_{i,j} = 0 /* ∀i, j */
5:   for j = 1 to m do
6:     Call FindMaxMinVM() to get MaxGain_j, MinLoss_j, VM_l, VM_g
7:     Call CompareGainAndLoss()
8:     CheckReshuffle() /* Alter, if necessary, the resource winner and/or loser VMs by checking against multiple increments/decrements */
9:   if MaxNetProfit > 0 then
10:    R_{VM_gg, R_max} += δ
11:    IC_{VM_gg, R_max} += 1
12:    R_{VM_lg, R_max} -= δ
13:    DC_{VM_lg, R_max} += 1
14:  else
15:    /* No gain in net profit and the algorithm stops */
16:    break
```

Algorithm 2 $FindMaxMinVM$: Find VMs with maximum and minimum gain respectively

```
1: OUTPUT: MaxGain_j, MinLoss_j, VM_l, VM_g
2: MaxGain_j = 0
3: MinLoss_j = ∞
4: for i = 1 to n do
5:   /* Finding the VM whose gain is maximum */
6:   if δP^g_{i,j} > MaxGain_j and IC_{i,j} < k then
7:     MaxGain_j = δP^g_{i,j}
8:     VM_g = i
9:   /* Finding the VM whose loss is minimum */
10:  if δP^l_{i,j} < MinLoss_j and DC_{i,j} < k then
11:    MinLoss_j = δP^l_{i,j}
12:    VM_l = i
13: return MaxGain_j, MinLoss_j, VM_l, VM_g
```

Algorithm 3 $CompareGainAndLoss$

```
1: INPUT: MaxGain_j, MinLoss_j, MaxNetProfit, VM_g, VM_l
2: OUTPUT: VM_gg, VM_lg, R_max
3: if VM_g ≠ VM_l and MaxGain_j + MinLoss_j > MaxNetProfit then
4:   MaxNetProfit = MaxGain_j + MinLoss_j
5:   VM_gg = VM_g
6:   VM_lg = VM_l
7:   R_max = j
```

The main algorithm (`MaxRevenue` Algorithm 1) implements an incremental reallocation of resources across application VMs. This algorithm is run each time a resource redistribution across VMs is considered; this could be either periodic or based on administrator initiation. The algorithm `MaxRevenue` identifies the VMs offering the maximum gain and minimum loss for δ change of all resource types j (the

Parameter	Description
i	Index for application VMs
j	Index for resource types
$\delta P^g_{i,j}$	Gain in revenue for application VM i as resource allocation of type j is increased by δ keeping all other resource dimensions constant
$\delta P^l_{i,j}$	Loss of revenue for application VM i as resource allocation of type j is reduced by δ keeping all other resource dimensions constant
$MaxGain_j$	Maximum Gain obtained for resource type j
$MinLoss_j$	Minimum Loss incurred for resource type j
MaxNetProfit	Maximum net profit obtained globally i.e. across all VMs and all resource dimensions
VM_g	VM whose gain is maximum for a specific j
VM_l	VM whose loss is minimum for a specific j
VM_{gg}	VM whose gain is maximum for some j and which is globally selected as a candidate for allocating more resources
VM_{lg}	VM whose loss is minimum for some j and which is globally selected as a victim for taking away resources
R_{max}	Resource type for which the net profit is maximized
$IC_{i,j}$	Number of times the resource allocation of type j for VM i is incremented
$DC_{i,j}$	Number of times the resource allocation of type j for VM i is decremented

TABLE II. DESCRIPTION OF SYMBOLS USED IN THE HEURISTIC SOLUTION.

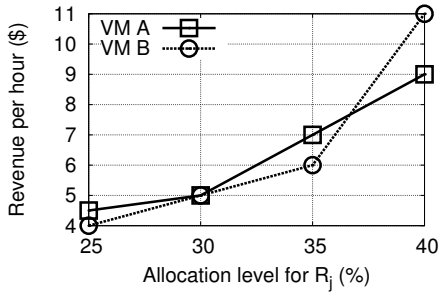


Fig. 2. Sub-optimal allocation with multiple δ increments of resource R_j.

`for` loop at line 5 of Algorithm 1). In line 6, the algorithm invokes `FindMaxMinVM` (Algorithm 2) to identify the VM that offers the minimum loss of revenue due to loss of δ amount of resource j and the VM that provides the maximum gain in revenue for the addition of the same amount.

Our approach is based on making potentially multiple changes to resource allocation across multiple iterations, with only a small, incremental (δ) resource allocation change within a single iteration. This enables the algorithm to partition resources at a fine granularity and consider a large number of resource reallocation possibilities. This also allows the redistribution of a resource from a single donor VM to multiple recipient VMs and from multiple donor VMs to a single recipient VM. However, one disadvantage is that the allocation result achieved at each iteration due to a δ change may not be cumulatively optimal, i.e., for multiple δ change. This is illustrated in Figure 2 which depicts the change in revenue for two hypothetical VMs A and B as their allocation for resource R_j changes. Let us assume that the current allocation level of R_j for both VM A and VM B are 30% which generates a revenue of 5$/hr for both VMs. Let us further assume for simplicity that δ=5 and k=2; these values will typically be different in a real setting with k being greater and δ being either larger or smaller depending on the accuracy of the model w.r.t. modeling the impact of the specific resource. During its next execution, the algorithm would determine in the very first iteration that VM A offers a greater increase in revenue (2$/hr) for a δ (5%) increment in R_j allocation from 30% to 35% than VM B which offers a lower increase (1$/hr) for the same increment. In the second iteration, once again VM A offers a greater increment (2$/hr) for an increment from 35% to 40%, while VM B offers only (1$/hr) for an increment of 5% from 30% to 35% of R_j. However, if we make the allocation granularity more coarse grained in the first iteration (say 2δ) then the 10% allocation recipient would have been VM B which offers a greater cumulative increase in revenue (5$/hr) as opposed to VM A (4$/hr).

The `CheckReshuffle` function addresses the above shortcoming. It compares the sum of all changes determined as piece-wise optimal in previous iterations with the entire reallocation made as a single unit made at once (i.e., effectively increasing the size of the allocation unit). If `CheckReshuffle` establishes that the larger granularity allocation of a single resource is more beneficial than incremental δ reallocations, it modifies the VMs assigned for maximum gain and minimum loss during the current iteration.

In its final section, (lines 9-16), the Algorithm 1 checks if the *MaxNetProfit* is greater than 0, i.e., there exists an additional revenue benefit from resource redistribution. Upon success, the resource transfers and other manipulations occur from lines 10 to 13 and the outer while loop is executed once again. Otherwise, the algorithm is unsuccessful in finding a better resource assignment than what was identified during the last iteration and it stops by breaking from the outermost while loop. The actual resource distribution only occurs after the algorithm finishes its entire execution, i.e., after having identified potentially multiple source and target VMs and multiple types for resources.

V. Implementation

We developed our system based on VMware's virtualization stack for servers and server management. For building performance models, VM workloads were first run in a staging host identical to the target ESX server used for deploying the VMs. Performance models are recorded in a *central resource allocator* (CRA), a dedicated machine in our implementation. The CRA runs the resource allocation algorithm and provides reallocation hints. Application VMs running on the staging host and target ESX server access virtual disks in a network storage system over NFS.

Performance data from individual virtual machines are collected and transmitted to the CRA. VM applications need to report this information that is consumed by our system either

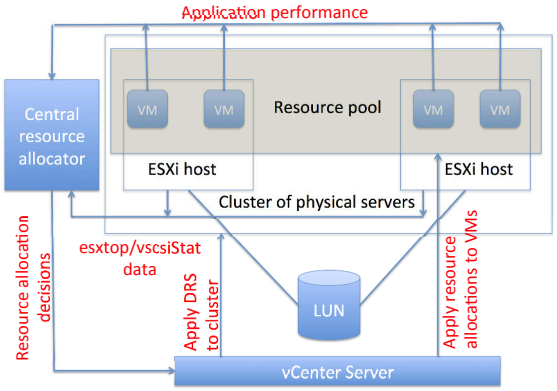


Fig. 3. Cluster-wide revenue driven dynamic resource allocation setup.

through files or console. We found that this was straightforward for the benchmark workloads we used since they reported performance data to file/console by default. Each ESX host runs the *esxtop* tool to collect per-VM level as well as host-level performance data. We used vscsiStats [4] on each ESX host which reports storage I/O latency statistics for each VMDK. These are transmitted to the CRA, which in turn makes resource reallocation decisions every five minutes.

The resource allocation process is shown in Figure 3. Initially, the available CPU and memory in the target server are divided among the running VMs either equally or in proportion to the application-specified SLA weights or priority values. VMs continuously report the application performance to the CRA. After a five minute epoch elapses at the CRA, the SLA functions are consulted to transform the observed application performance to corresponding revenue values in USD. The revenue maximization algorithm is then run using the model-predicted revenue data and a new set of resource assignments for the pool of VMs are generated. The new assignments are informed to the vCenter Server which implements the new allocations. The VMs run for another five-minute epoch and the whole process repeats.

A. Resource Assignment Mechanisms

An ESX host provides several control knobs for managing resource assignments to individual VMs. Irrespective of the resource assignment mechanism (i.e., limits or shares), dynamic resource allocation works as discussed earlier. *Limit*, *reservation*, and *share* can each be used to control the allocation of CPU and memory to VMs [11]. *Limit* places an upper bound on the amount of resource a VM can consume; *reservation* guarantees a certain minimum amount of resource to a VM at any time. *Share* allows ESX to dynamically vary the resource allocated to a VM between its reservation and limit values, in proportion to specified priority values and based on actual demand. The work-conserving nature of shares makes it attractive for resource distribution. On the other hand, while limits are non-work-conserving, they provide strict resource isolation across the VMs.

While *shares* and *limits* provide useful controls, it is unclear how per-VM limits or shares should be configured to maximize revenue. Our dynamic resource allocation solution addresses this gap. Although our algorithm is intended to use resource *limits*, we adapted it to use *shares* as well by mapping each suggested limit to a corresponding share value by dividing the resource assignment with the total capacity of the resource. Thus, we were able to evaluate four distinct schemes: *(i) Share_Reservation:* shares are used in combination with some reservation for both CPU and memory; *(ii) Share_noReservation:* shares are used without any reservation for both CPU and memory; *(iii) Limit_Reservation:* limits are used with reservations for both CPU and memory; and *(iv) Limit_noReservation:* limits are used without any specified reservations.

B. Cluster-wide Scaling

Our solution for dynamic resource management works within a single server by observing and utilizing the fine-grained impact of resource reallocation decisions. However, our solution can be applied at cluster scale by employing the ”resource pool” abstraction in virtualized cluster management solutions such as the vCenter Server (see Figure 3). The VMs are administered using VMware vCenter Server [5]. The concept of *resource pool* restricts allocations to the resource pool but a resource pool could include either a single ESX host or a group of ESX hosts. It can thus provide straightforward application of our resource management mechanisms to multiple hosts [11]. VMs are placed on the resource pool with a condition that the sum of reservations on any resource dimension to the pool of VMs is not to exceed the reservation on the pool.

The advantage of using resource pools is that it aggregates physical resources from multiple hosts creating the illusion of a single virtual server. In other words, a resource pool is constructed using a cluster and the cluster in turn is comprised of multiple physical machines. This resource virtualization achieves transparent migration of the pooled VMs between the hosts in the cluster at run-time. Migration may occur either as a result of a certain allocation assignment to a particular VM being deemed unsupportable by the current server or due to internal load balancing operations [11].

VI. Evaluation

In this section, we demonstrate how starting with an initial configuration VM resource assignments, at each iteration, changes to resource assignments that ultimately converge to an increase in total revenue for the data center can be realized. To evaluate the effectiveness of the proposed revenue driven dynamic resource allocation framework, we compare it against solutions that use existing mechanisms in the ESX hypervisor for dividing allocations based on relative VM priority, e.g., VM *shares*.

A. Testbed and Workloads

We used an AMD-based Dell PowerEdge 2970 server with dual sockets and six 2.4 GHz cores per socket and 32 GB of physical memory, running the VMware ESXi- 5.1 hypervisor, as the target host system to run virtual machines. All the VMs ran Ubuntu-Linux-10.04. The virtual machine disks (VMDKs) were placed in a 1.2 TB RAID-0 LUN using four disk drives hosted by a networked storage server. ESX mounted these VMDKs at the host using NFS. The *central resource allocator* was run on a dedicated Dell PowerEdge T105 machine with

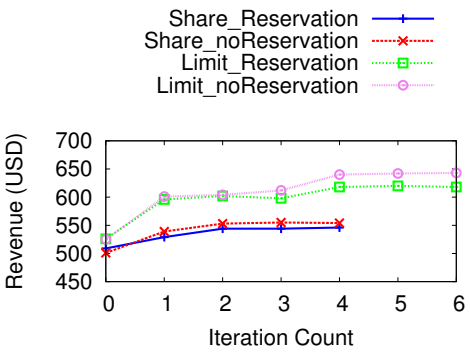


Fig. 4. Incremental change in revenue when started with equal resource allocations. Resource reallocation iterates until revenue generation stabilizes.

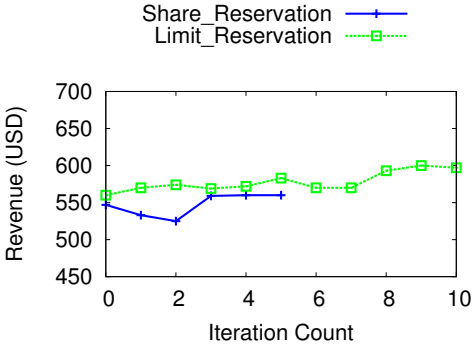


Fig. 5. Incremental change in revenue when the initial resource allocations to VMs are assigned proportionally to respective SLA weights. Resource reallocation iterates until revenue generation stabilizes.

a quad-core AMD Opteron processor (1.15GHz×4), 8 GB of physical memory, and a 7.2k RPM disk running Ubuntu-Linux-10.10.

For the workloads, we used two profiles each of RUBiS [3] (Browsing and Bidding) and Filebench [2] (Webserver and Fileserver); these workloads consume multiple physical resources (CPU, Memory and I/O) at the same time in a complex manner [16]. We configured the RUBiS-Browsing workload to 1000 clients and the RUBiS-Bidding workload to run 400 clients simultaneously. Both reported average throughput as requests/sec, which was used as the application performance metric when evaluating candidate solutions. We configured the Filebench Webserver and Fileserver workloads each with a fileset of total size 10GB and to use 32 threads. Application performance for both was recorded as operations/sec.

B. System Configuration

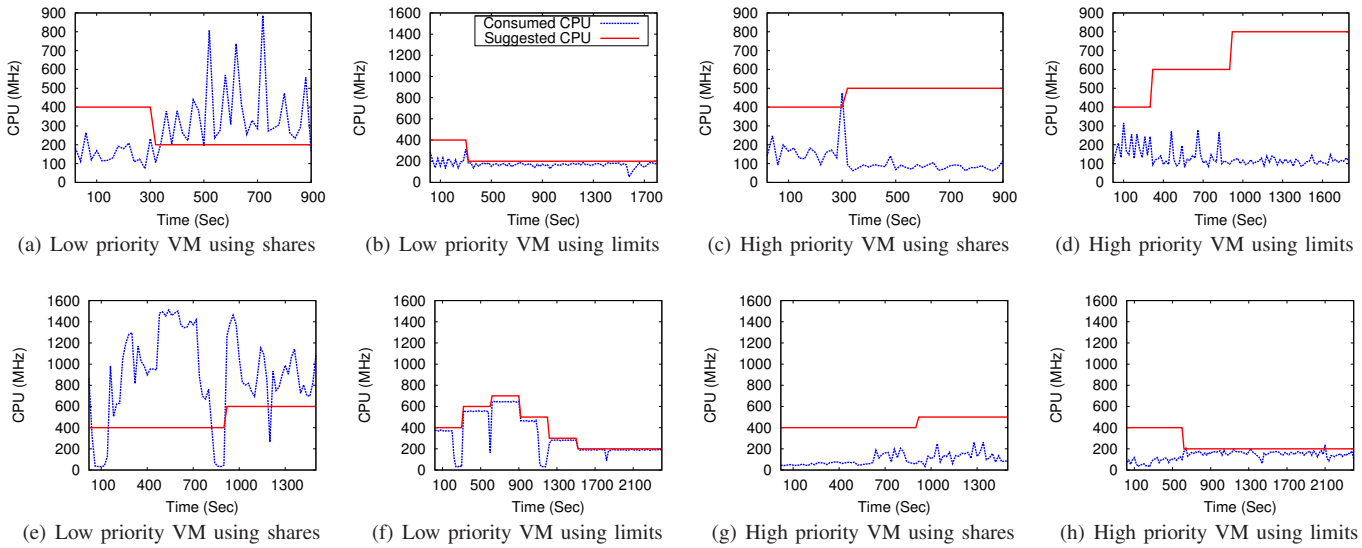
We trained models with data points separated from each other in the parameter space at coarse granularity as described in our previous work [16]. Choice of δ, the smallest granularity of resource movement, proved to be an important one. From preliminary experiments, we found that choosing a very small value lead to prediction inaccuracy. On the other hand, since the performance models use the current observed VM storage I/O latency to predict the application performance for next iteration, it is important that the stability of I/O latency is ensured when resource changes are made. Thus, the combined value, kδ — the maximum resource change suggested by the resource allocation algorithm, should be bounded to ensure the stability of the VM storage I/O latency. We empirically chose δ, to be 100 MHz for CPU and 64 MB for memory, both of which worked well for the RUBiS and Filebench workloads we evaluated with. Similarly, we chose the value of k as 2. In other words, in each epoch of the resource allocation procedure, a VM was allowed to have a maximum change of 200 MHz of CPU and 128 MB of memory from its previous assignment.

When using *reservations*, we set each VM to have at least 200 MHz CPU and 256 MB of memory. For configurations that did not use reservations, these values were set at zero. When using *shares*, we set the limit of each VM to the capacity of the target *resource pool* thereby forcing SLA-based prioritization in resource partitioning. When using *limits*, all the VMs were initialized with equal shares. Further, the sum of per-resource *limits* of the VMs was set to be equal to the capacity of the resource pool. SLAs were chosen as simple weight values to transform the normalized application performance metrics to US Dollars. While more complex SLAs are often employed in practice, these SLAs only affect the revenue generation mechanism within our framework and not the performance prediction accuracy. Consequently, we anticipate the findings with more complex SLAs would be similar when our solution is compared to alternate solutions that do not model the impact of resource allocation on revenue generated.

C. Benefits of Isolation

To evaluate the utility of limit-based resource isolation, we used 10 VMs each running an identical instance of the Filebench Webserver [2] workload. Since our servers were large multi-core systems with large amounts of memory, for this experiment, we created a resource pool in the target server to restrict the total available resource capacity for these VMs to simulate the contention effects seen under high consolidation. The capacity of the resource pool was capped at 4 GHz for CPU and 4GB for memory; these values were chosen based on our preliminary measurements of in-memory working sets of the workload so that there is moderate storage I/O activity per-VM under equal allocation. The performance obtained from each VM is mapped to revenue in USD as *Revenue* = *Performance* × *SLA weight*, where *SLA weight* of each VM is assigned a value between 1 to 6.

Our experiment was initialized with resources being distributed equally among the VMs. Figure 4 shows how the total revenue from 10 functionally identical filebench webserver VMs changes with our algorithm after each iterations separated by five minutes. We compare the four resource assignment mechanisms (described in §V-A) when they are combined with our dynamic resource allocation solution. Successive iterations monotonically drive the system from a state of lower revenue to a state of higher revenue. *Limit_noReservation* provides a 22% increase in revenue with respect to the initial placement. *Share_noReservation* provides a 10% revenue increment. Overall, however, the limit-based approach provides much higher revenue (18% higher) than the share-based approach. This result highlights the utility of isolating assignments in combination with the greedy heuristic to automatically increase the revenue of virtualized data centers. Starting from a different initial configuration where resources are assigned proportionally to the application-specified SLA weights, we observed a



(a) Low priority VM using shares (b) Low priority VM using limits (c) High priority VM using shares (d) High priority VM using limits

(e) Low priority VM using shares (f) Low priority VM using limits (g) High priority VM using shares (h) High priority VM using limits

Fig. 6. Actual CPU consumption versus suggested CPU allocation for two VMs across multiple iterations of the resource allocation algorithm. The chosen VMs have the lowest and the highest SLA weights respectively. Suggested allocations are applied to the VMs using either *shares* or *limits*. The upper set of subfigures (a, b, c, and d) represent the small scale experiment and the lower set of subfigures (e, f, g, and h) represent the large-scale 25 VM experiment.

gain of 7% (Figure 5). Here too, using limits provided higher revenue than using shares.

To understand why using work-conserving shares led to relatively poor outcome when compared to using isolating limits, we analyzed the CPU utilization over time of the VMs with the lowest and the highest SLA weights respectively (Figures 6(a)-(d)). We compared these with the CPU allocation suggested by our algorithm. We see that shares allow the allocations of the VMs to fluctuate arbitrarily irrespective of the suggested assignments as demand varies. Despite the low priority VM achieving higher performance, it did so by transitorily reducing the allocations of the higher priority (and thus higher revenue generating) VM which ultimately affected total revenue negatively. Although per-VM shares were tailored to match their respective SLA weights, the functionally identical nature of the VMs left little room for the hypervisors to use up transitorily under-utilized resources without negatively affecting the performance of higher priority VMs. Moreover, the similar workloads are similarly CPU, memory, and I/O bound and increase the contention for shared resources, thereby motivating isolation-based implementation of prioritization. Thus, limit-based allocation which enforces such isolation delivers much higher total from multiple hosted VMs. In both allocation methods, the overall CPU utilization of the cluster were equal or lower than the configured CPU capacity of the resource pool.

D. Benefits of Work Conserving Behavior

As a next step, we wanted to understand the effects of resource limits and shares when VM activity is more diverse. We ran a larger-scale experiment with 25 VMs with a diverse mix of 7 filebench-webserver VMs, 6 filebench-fileserver VMs, 6 RUBiS-Browsing and 6 RUBiS-Bidding VMs. The CPU and memory capacity of the resource pool were set at 10 GHz and 10 GB respectively and the initial configuration set up VMs

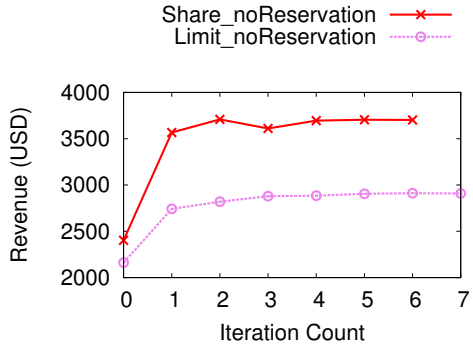


Fig. 7. Large-scale experiment with 25 VMs with dissimilar workload types. Resource reallocation iterates until revenue generation stabilizes.

with equal allocations. As before, the *SLA weight* of each VM is assigned a value between 1 to 7.

We start the system with an equal allocation of resources across VMs. Figure 7 reports a 34% increase in revenue compared to initially observed revenue when using *Limit_noReservation* and 54% increase when using *Share_noReservation*. Moreover, the final revenue state obtained using shares is 27% more than when using limits, quite unlike our observations with similarly behaving VMs.

To investigate why shares delivered higher revenue in this experiment, we performed a comparative analysis of CPU consumption. We chose a filebench-webserver VM and a RUBiS-browsing VM with higher and lower SLA weights respectively. Figures 6 (e)-(h) reveal that actual CPU consumed by the VM with higher SLA weight while using limits is often less than the allocated amount. Further, we note that the CPU bound RUBiS browsing VM utilizes the unused CPU cycles while using shares even though it has lower SLA weight. These observations suggest that filebench-webserver is not CPU demanding during the entire runtime. Since the pool of

VMs in this experiment is diverse, they are less likely to be similarly resource bound. Since limit is not work-conserving, the unused CPU cycles cannot be used by any other CPU bound VM (e.g. RUBiS-browsing VM). Share, being work-conserving, can reallocate the unused CPU cycles to another VM as necessary which helps increase the total revenue.

When using shares, the total CPU utilization of the cluster matched the capacity of the resource pool. When using limits, the average total CPU utilization of the cluster was 90% of the capacity of the resource pool. When we repeated the 25 VMs experiment with initial shares assignment proportional to VM SLA weights, the total revenue increased by 29% compared to initial revenue. But, we did not see any significant gain from the initial total revenue using limit based allocation process when initial resource assignments were based on SLA weights. The final revenue state achieved by using shares was 30% higher than that achieved by limits. These findings underscore the benefits of work conservation when running diverse workloads.

E. Summary of Results

Isolation based allocations provide higher revenue when workloads are similarly resource consuming and the total resource pool capacity is limited. On the other hand, work-conserving allocation mechanisms deliver substantial gain when the total resource pool capacity is large and the workload mix is diverse. These results highlight the effectiveness of revenue-based dynamic resource allocation system, which irrespective of the underlying control knobs (share or limit), drives the data center revenue to a significantly higher level.

VII. Related Work

Previous work on resource allocation in virtualized environments falls into two broad categories: (a) application-specific solutions that employ domain knowledge, and (b) dynamic reallocation of a specific resource type.

A. Application-specific Solutions

Aboulnaga et al. [22] proposed automatic virtual machine configuration for database workloads. Their virtualization design advisor uses application-specific information about the database workloads to reduce the execution cost for each VM hosting a database instance. Dutta *et al.* optimized storage I/O utilization by monitoring the usage of various database elements [9]. In contrast to these works that rely on expert knowledge about the database, our approach is application-agnostic and relies only on coarse-grained performance data reported by the application. ActiveSLA [24] suggested a framework for admission control of individual queries in cloud database systems where admission decisions are guided by SLAs and expected profits. In contrast, we formally model the application-independent version of the dynamic resource reallocation problem and compare it against known allocation-based optimization problems. We also develop and evaluate an efficient resource allocation algorithm that provides a widely applicable heuristic solution.

B. Resource-specific Solutions

Recent work on adaptive control of virtualized resources in data centers [19] describes an approach for handling multi-tier applications with the high-level goals of *(i)* guaranteed application-level QoS, *(ii)* high-resource utilization across all physical nodes, and *(iii)* QoS differentiation during resource contention. However, this previous work only considers CPU allocation and it is not clear how effective the proposed approach would be when used to manage other resource types.

CloudScale [21] is an automatic elastic resource scaling system for multi-tenant cloud services. It is designed to minimize Service Level Objective (SLO) violations while optimizing physical resource and energy usage. Unlike CloudScale which only addresses CPU demand, we demonstrate and account for the fact that several applications depend on multiple resource types for achieving their performance objective [16]. Salomie et al. [20] proposed application level ballooning (ALB) that extended the conventional memory ballooning technique to better control applications that manage their own memory. In Ginkgo [10], authors utilize application level and system level data to measure memory requirements and distribute memory across virtualized applications at run-time to achieve administrator defined objectives. Ben-Yehuda et al. [6] developed a bidding system (Ginseng) where memory is awarded or deducted based on the willingness of the customer VM to pay for contested memory. ALB, Gingko, and Ginseng, all being memory allocation and controlling mechanisms, focus on managing one resource for performance control. In this work, we address the problem of partitioning multiple resource types that impact application performance for data center revenue maximization using a unified solution that can account for an application's dependence across multiple resource types.

Pesto [12] and *Cake* [23] focus on storage resource provisioning to meet administrator-specified storage SLOs. Xu *et al.*'s work [25] considered the profit-driven resource optimization problem and formalized it as an instance of the continuous knapsack problem which they solved using a greedy allocation algorithm. In comparison, our work recognizes and illustrates the discrete nature of VM resource allocation and the need for incremental resource reallocation in order to ensure system stability and performance prediction accuracy.

Q-cloud [17] aims to mitigate performance interference caused by co-located VMs so that clients get the same performance as will be achieved by running the application in a dedicated system. The models used in Q-cloud assume a degree of linearity in resource consumption vs. performance. Subsequently, however, we have shown that application performance depends on resource availability in a non-linear fashion [16]. The most important difference of our work lies in our goals. We are not simply interested in performance interference removal but also envision a market or revenue driven approach where application VMs are rewarded or penalized based on their respective performance-based SLAs.

Recently, Bryant et al. proposed a micro-elastic server called *Kaleidoscope* [8] to dynamically create worker VM clones to satisfy the increased demand in a target VM. However, this work does not address how the physical resources should be distributed to VMs according to their respective SLAs. We view Kaleidoscope as a complementary solution that

satisfies instantaneous load spikes in user VMs. On the other hand, our revenue driven approach delivers effective resource partitioning when the loads on the VMs are stable and the resource allocation decision is guided by the SLAs.

Finally, autoControl [18] addresses: *(i)* CPU as well as disk I/O resources, *(ii)* service level objectives within the contention differentiation metric, and *(iii)* the restriction of hosting a particular tier in a specific node. Our work differs from autoControl in our handling of memory resources and in our explicit objective of total revenue maximization across a data center whereby we address the fundamental tension between the cloud service provider and client.

VIII. Conclusions

Optimal management of data center resources is a crucial yet cumbersome task. While promising to relieve administrative complexity, virtualization has also compounded the resource management problem by enabling high degrees of workload consolidation within a single host. In this paper, we designed, built, and evaluated a novel revenue driven dynamic resource allocation solution which partitions the available physical resources among a pool of VMs with the goal of attaining high SLA-based revenue for the data center operators. Performance models were integrated with a hill-climbing algorithm that performs incremental resource reallocation to achieve the objective of maximizing SLA-based revenue. We demonstrated that our solution can be built upon the existing mechanisms for resource assignment such as resource shares, limits, and reservations in a straightforward way. We evaluated our approach using an ESX host and a networked storage server hosting the virtual disks, demonstrating that both resource isolating limits and work-conserving shares are relevant depending on whether VM behavior is homogeneous across the host. Our results indicate that the proposed dynamic resource allocation solution drives the system to substantially higher revenue levels in comparison to static partitioning of the available resources as well as when per-VM shares are configured in proportion to SLA weights.

Acknowledgments

We thank the anonymous reviewers for their feedback which helped improve the material presented in this paper. This work was supported in part by NSF awards CNS-1253944 and CNS-1018262. Sajib Kundu was also supported by an FIU Dissertation Year Fellowship.

References

[1] Amazon elastic compute cloud (amazon EC2). http://aws.amazon.com/ec2/.

[2] *Filebench: a framework for simulating applications on file systems*. http://www.solarisinternals.com/wiki/index.php/FileBench.

[3] *RUBiS: Rice University Bidding System*. http://rubis.ow2.org/.

[4] *Using vscsiStats for Storage Performance Analysis*. http://communities.vmware.com/docs/DOC-10095.

[5] Vmware vcenter server. http://www.vmware.com/products/vcenter-server/.

[6] O. A. Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on virtual execution environments (VEE)*, pages 41–52, 2014.

[7] K. M. Bretthauer and B. Shetty. The nonlinear knapsack problem – algorithms and applications. *European Journal of Operational Research*, (138):459–472, 2002.

[8] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, H. A. Lagar-Cavilla, and E. D. Lara. Kaleidoscope : Cloud micro-elasticity via vm state coloring. In *Proceedings of the sixth conference on Computer systems (EuroSys)*, pages 273–286, 2011.

[9] K. Dutta, R. Rangaswami, and S. Kundu. Workload-based generation of administrator hints for optimizing database storage utilization. *Trans. Storage*, 3(4), Feb. 2008.

[10] A. Gordon, M. R. Hines, D. da Silva, M. Ben-Yehuda, M. Silva, and G. Lizarraga. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE) Workshop*, 2011.

[11] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. *VMware Distributed Resource Management: Design, Implementation and Lessons Learned*. http://labs.vmware.com/publications/gulati-vmtj-spring2012, 2012.

[12] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[13] P. L. Hammer. *Studies in integer programming*. IBM Deutschland.

[14] R. Koller, A. Verma, and R. Rangaswami. Generalized ERSS tree model: Revisiting working sets. *Performance Evaluation*, 67(11):1139–1154, 2010.

[15] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao. Application Performance Modeling in a Virtualized Environment. In *Proc. of IEEE High Performance Computer Architecture (HPCA)*, January 2010.

[16] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta. Modeling Virtualized Applications using Machine Learning Techniques. In *Proceedings of the 8th ACM conference on Virtual Execution Environments (VEE)*, March 2012.

[17] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys '10*, pages 237–250, 2010.

[18] P. Padala, K.-Y. Hou, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. G. Shin. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems/EuroSys*, pages 13–16, 2009.

[19] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proc. of Eurosys*, pages 289–302, 2007.

[20] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 337–350, 2013.

[21] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, 2011.

[22] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 953–966, 2008.

[23] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level slos on shared storage systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC)*, 2012.

[24] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacigumus. Activesla: A prot-oriented admission control framework for database-as-a-service providers. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, 2011.

[25] J. Xu, M. Zhao, J. A. B. Fortes, R. Carpenter, and M. S. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, 2008.