Towards Intelligent Management in VM-based Resource Providers *

Fernando Rodríguez, Felix Freitag, and Leandro Navarro

Polytechnic University of Catalonia, Computer Architecture Department, Jordi Girona, 1-3, D6 Campus Nord, 08034 Barcelona, Spain,
frrodrigu@ac.upc.edu

Abstract. In this paper, we present and evaluate an algorithm to provide autonomous management in VM-based resource providers. We provide a solution to adjust the resources needed by each VM according to demand. We achieve this with a policy that meets at least the agreed application-level QoS, and a priority weight-based mechanism to spread the unused resources between the contending VMs that run over their QoS requirements. The proposed algorithm is implemented in a two processors node with Xen virtualization solution and credit scheduler. The results show that we can better manage physical resources by dynamically adjusting local resources according to demand.

1 Introduction

The recent advances in virtualization [1][2] in conjunction with the advent of symmetric multiprocessing (SMP) architectures, faster processors and faster memory access in desktop computers have attracted the attention of the Grid research community [3][4][5]. Moreover, works in the research area of High Performance Computing have started to explore virtualization as a promising area [6] to provide CPU cycles for solving intensive computing problems [7].

One of the key issues interesting for these research communities is the local management of VM-based resource provisioning.

Traditionally, without virtualization, the problem of resources provisioning and management has been tackled using a pool of physical nodes that are tied to an specific operating system. This scenario leads to high infrastructure and management costs, but more important, the infrastructure is not efficiently used due to the waste of resources in the under-utilized physical node (i.e. an application could not need the total of resources provided in the physical node and under-utilized some of the physical resources like CPU or network).

Unlike traditional approach, virtualization makes possible to boot different isolated Operating Systems sharing the multiplexed physical resources of a node. The isolation is achieved through the concept of the Virtual Machine (VM),

* This work is supported in part by the European Union under Contracts SORMA EU IST-FP6-034286 and CATNETS EU IST-FP6-003769, the Ministry of Education and Science of Spain under Contract TIN2006-5614-C03-01, and the program PROMEP.
which is a virtual extension of the physical machine’s resources. The Virtual Machine Monitor (VMM) is the software that supports virtualization [8].

With the aid of VMs, we can consolidate in a physical node the workloads of different applications. However, the virtualization layer is not aware of the application-level requirements. Hence, it can not determine the amount of resources needed by each application. For these reasons, we have identified the need of dynamic solutions to conciliate and resolve the workload requirements of the running applications in the context of VM-based resource providers.

In this paper we present and evaluate an algorithm to provide autonomous management in VM-based resource providers. The solution adjust the resources needed by each VM according to demand. The algorithm implements a policy that meets at least the agreed application-level Quality of Service (QoS) during the VM creation, additionally, it implements a priority weight-based mechanism to spread the unused resources between the contending VMs that run over their QoS requirements.

This paper is organized as follows: In section 2 we describe work related to our approach. In section 3 we explain the management component and the sensor and controller algorithms. In section 4 we report on several experiments in which we study the implemented component. Section 5 concludes and section 6 provides an outlook on future work.

2 Related work

Dongyan Xu et al [9] proposes the concept of virtual distributed environments (VDE) as a new sharing paradigm for multi-domain shared infrastructure. Authors also propose to support autonomic adaptation of virtual distributed environments, driven by both dynamic availability of infrastructure resources and dynamic application resource demand. A centralized adaptation software manages the infrastructure nodes and adjust their resources shares and locations. The difference is that our approach is decentralized and focus on intra LRM fine-grain adaptation mechanisms.

Padala et al [10] propose an adaptive control system based in classic control theory. The purpose of the controller is to alleviate the poor utilization in traditional data centers. The experimental setup includes virtualization and multi-tier applications and the results obtained shows that the controller approach meet application-level QoS goals and the data center achieve high utilization. However, the controller used only static models and some experiments could not been performed due to some anomalies that the authors found in the Xen simple Earliest Deadline First (sEDF) scheduler.

We propose a Multiple Dimension Slotting Approach (MuDiS) [11] to offer a solution for fine-grain management in the framework of a VM-based resource provider. The solution adds a partitioning layer between the low-level virtualization layer and the local resource manager, with this is possible to partition an SMP node and to manage the proportion of each processor in fine-grain detail. Each partition may then be assigned to certain type of applications or used to
reallocate VMs. However, the implemented prototype offers static partitioning and manages the CPU resources through the Xen sEDF scheduler and the Xen pinning function.

The reviewed works support adaptation mechanisms. They are in the direction to provide solutions in order to adjust appropriately the needed resources to a new VM without violating the ones that were previously agreed. Our work is in the direction of the first two works, specially we aim to solve adaptation problems within the physical nodes.

3 Management of VM-based resource providers

The controller algorithm for the management of VM-based resource providers is implemented in the python language in a two processors node with Xen virtualization solution and credit scheduler. The preliminary results shows that we can better manage physical resources by meeting the QoS requirements of all VMs under heavy load conditions while allowing the dynamic adjusting when unused resources are sensed.

3.1 Execution context

We have implemented a Local Resource Manager (LRM) for the VM-based resource provider. The VM creation request includes a job description requirement. We set the memory requirement to a fixed value of 192MB. Concerning to the CPU resource the following information must be provided:

- An application-level requirement expressed in MHz and
- A differentiation service type which can be fixed, variableLow, variableMedium or variableHigh

This information, received from the client requests, represents the high level input parameters. A process maps these values to low level objectives in order to slice properly the resources managed by the virtualization layer. This approach aims to assure two objectives: first, exposes certainty about the agreed performance to external management agents such as global schedulers, and second, allows to aggregate resources to applications with different time-varying workloads requirements.

With this information the LRM accepts or rejects the VM creation. If the creation is accepted then we add two fields in the VM entry of the XenStore database: QoS with the MHz expressed in percentage of the node capacity and \( P \) with the value 0 for fixed, 2 for variableLow, 4 for variableMedium and 8 for variableHigh.

3.2 Management component

The management component includes the sensor algorithm and the controller algorithm. The algorithms run in parallel and communicate each other by a queue.
The algorithms are scheduled to be called at different time periods (sensorPeriod and controllerPeriod).

The sensor algorithm calls the VMMsensor primitive of the virtualization layer to collect the CPU metrics and compute the utilization of each VM (the utilization varies from 0 to capacity \(100 \times \text{numProcessors}\)), finally the sampled metrics are put in the sensor queue \(S\).

The algorithm 1 shows the steps for the controller. This algorithm is called every controllerPeriod (sensorPeriod \(\times \text{numberSamples}\)). From lines 5-11 we empty the sensor queue \(S\) and account metrics for each VM in \(U\) and \(t\), the utilization mean for each VM is computed in line 13.

Lines 15-16 solves the case when the utilization of a VM runs under its agreed QoS; for the privileged VM we follow a policy to assign a reservation \(R\) of agreed QoS, for the rest of the VMs we assign a reservation \(R\) of at least 5% of the capacity. We choose the value 5% to assure an appropriate performance in the VM in case it becomes idle and tries to re initiate simple tasks such as remote secure shell among others. Lines 18-20 solve the case when a VM runs over its agreed QoS; the exceeding utilization of the VM is stored in contention and the reservation \(R\) is set to QoS. 

In Line 23 we account the available resources, note that this value does not correspond to the idle system (see Lines 15-16). In line 25 we apply proportional share (PS) to build the allocable vector of the contending VMs. In line 28 we increase the reservation of the VMs that run over its QoS to the minor value of its exceeding utilization and allocable share. With the computed reservations \(R\) we apply a correction function to obtain the reservation (cap) for the next period. In Line 37 we make a sanity check for the cap value in case it surpass the VM capacity. Finally the algorithm calls the VMMactuator primitive of the virtualization layer with the cap value.

4 Experiments and results

In the experiments the hardware of the resource provider node has a Pentium D 3.00GHz (two processors), 1Gb of RAM, one hard disk (160GB, 7200 rpm, SATA), and one network interface card (Broadcom NetXtreme Gigabit). The operating system is Fedora Core 7 and for virtualization we use Xen 3.1.0. We deploy three paravirtualized VMs (pc1, pc2, and pc3) with Fedora Core 7 for the guest OS.

For the VMMactuator we use the credit scheduler, this scheduler accepts weights and cap to adjust the resources assigned to each VM. In the implementation of the algorithm we set the cap parameter with the computed value and leave the weight parameter to 256. As we are using an SMP node, the VMs remains unpinned and can use all the available processors.

In order to stress each VM we launch an script that behaves as a CPU intensive job (we launched it twice to assure that each VM was using both processors). The script burns CPU cycles with the following command.

```
#cat /dev/urandom | gzip -c > /dev/null
```
Algorithm 1: Controller

Input: previous cap $C$, sensor queue $S$, agreed QoS, priorities $P$, node

Output: $C$, resources adjusted according to demand

1. $capacity ← 100 \times numProcessors$

2. foreach $vm ∈ node$ do
   3. set $P', U[vm], t[vm], allocable[vm], contention[vm], R[vm]$ to 0;

4. end

5. while $S ≠ \emptyset$ do
   6. remove $s$ from the queue $S$
   7. for $vm ∈ s$ do
      9. $t[vm] ← t[vm] + 1$
   10. end

11. end

12. for $vm ∈ U$ do
   14. if $U[vm] \leq QoS[vm]$ then
      15. if $vm$ is privDomain then $R[vm] = QoS[vm]$
      16. else $R[vm] = max(U[vm], 5)$
   17. else
      18. $P' ← P' + P[vm]$
   21. end

22. end

23. $available ← capacity - sum(R)$

24. for $vm ∈ P$ do
   25. if $vm ∈ contention$ then $allocable[vm] ← P[vm] / P' \times available$
   26. end

27. for $vm ∈ contention$ do
   28. $R[vm] ← R[vm] + min(contention[vm], allocable[vm])$
   29. end

30. for $vm ∈ R$ do
   31. $cap ← R[vm]$
   32. if $U[vm] > cap \times 0.95$ then
      33. $ceil ← QoS[vm] + allocable[vm]$
      34. $delta ← (ceil - cap) / ceil + 1$
      35. $cap ← cap + delta$
   36. end

37. $cap ← min(cap, maxVCPUsvm) \times 100)$

38. if $C[vm] = \emptyset$ or $C[vm] \neq cap$ then
   39. VMMActuator($vm, cap$)
   40. $C[vm] ← cap$
   41. end

42. end
Three experiments are discussed in order to evaluate the proposed algorithm. In all of them we request the creation of three VMs with application-level requirements of 900MHz, 1800MHz, and 2700MHz for pc1, pc2, and pc3 respectively. The LRM maps the application-level requirements to the node capacity and adds the field QoS in the XenStore database with values 30, 60, and 90 that corresponds to each VM, the Domain-0 is set to a QoS value of 20.

The sensor period is set 2.5 seconds and the number of samples for the controller is set to 2 which led to a controller period of 5 secs. The length of the experiments is set to 100 sensor samples (i.e. 250 seconds).

- The first experiment is the baseline experiment, here we create pc1, pc2, and pc3 VMs and only the sensor is enabled to collect the utilization metrics. The VMMactuator function is called during the booting process of each VM, the QoS requirement is used for the cap parameter.

- In the second experiment we request the creation of the same VMs with the controller enabled, besides the same QoS parameters, we request a differentiation service of variableLow, variableMedium and variableHigh for pc1, pc2 and pc3 respectively.

- The third experiment is setup as in the second, except for the requested differentiation service which is set to variableHigh, variableMedium and variableLow for pc1, pc2 and pc3 respectively.

Figure 1 shows the CPU utilization for the experiment one. At the start of the experiment we follow a workload injection order of pc1, pc2, and pc3 separated by approximately 40 seconds. At the half of the sensed experiment we stop the running application in each VM in order to start a new workload injection order of pc3, pc2, and pc1.

We observe that the fixed assignment assures a maximum utilization defined by QoS, however much of the node capacity remains unused. For instance, from second 25 until the second 50 there are almost 150% of (discounting Domain-0 share) available CPU resources (100*2 processors). On the other hand at saturation points (e.g. second 110) the used CPU is close to 180% of the node capacity. Finally, note that the CPU utilization of Domain-0 is close to 5%.

Figure 2 shows the CPU utilization (VMname_s) and the computed cap value (VMname_c) for the VMMactuator. The workload injection order is pc1, pc2, and pc3. The VM pc1, which requested the priority variableLow, reaches an utilization of approximately 170% at second 60 (i.e. pc1 request the lowest differentiation service but it is the only one contending for the available resources). As soon as pc2 workload increases, the controller adjust the cap value for both VMs (i.e. at second 130 both VMs receives more that their QoS). From second 100 to 150 pc1 and pc2 run out of its QoS, so the controller compute the new caps by: (1) obtaining the available resources with available = capacity − ΣQoS[VMs] = 85, and (2) applying PS for the contending VMs to obtain the cap for pc1 = 30 + 85 * 0.33 = 58 and for pc2 = 60 + 85 * 0.66 = 116. Finally, when pc3 increases its workload, the algorithm ensures at least its agreed QoS (90% of the node capacity).
Fig. 1. Credit scheduler without the control algorithm. Fixed assignment of QoS

Fig. 2. Workload injection order: Pc1, Pc2, Pc3. Differentiation service: variableLow, variableMedium, and variableHigh for pc1, pc2, and pc3 respectively. CPU capacity of 200 (2 processors). The VM pc1 reaches an utilization ($pc1_s$) of approximately 170% even with variableLow differentiation service. If $pc1_s > 0.95 \times pc1_{ct-1}$ then we apply a correction function to obtain the current cap value ($pc1_{ct}$).
Figure 3 shows the CPU utilization for the experiment three. As in the second experiment, from second 0 to second 100, pc1 can use all available resources. From second 100 to 150 pc1 and pc2 run out of its QoS so the controller computes the new caps for the contending VMs. The algorithm applies the same approach as in the second experiment with the corresponding values for the current differentiation services: i.e. \( \text{available} = \text{capacity} - \sum QoS[VMs] = 85 \), pc1 \( \text{cap} = 30 + 85 \times 0.66 = 86 \) and pc2 \( \text{cap} = 60 + 85 \times 0.33 = 88 \).

![sensor and controller](image)

**Fig. 3.** Workload injection order: Pc1, Pc2, Pc3. Differentiation service: \( \text{variableHigh} \), \( \text{variableMedium} \), and \( \text{variableLow} \) for pc1, pc2, and pc3 respectively. At second 125 pc1, and pc2, are computed according to the requested differentiation services. Starting from \( t=200 \), the controller ensures the QoS. Even though Dom0 has a reservation of 20\%, it never runs out of resources (i.e. the controller algorithm incurs in negligible overhead)

## 5 Conclusions

We have presented and evaluated an algorithm to provide dynamic adjusting of CPU resources in a multiprocessor VM-based resource provider. By means of experiments we have shown that we achieve this goal by (1) assuring at least the requested application service (expressed in MHz) for each VM in the resource provider, and (2) increasing and decreasing the assigned resources (if available resources are sensed) according to demand. The dynamic adjusting is based on a differentiation service request which is mapped to a weight-based function.

Even though we reserved a QoS of 20\% for the Domain-0, it incurs in very low overhead. This means that the implemented management component, with the sensor and controller, causes a negligible overhead.
An experiment with a VM requesting the differentiation service fixed was not presented, however the algorithm can handle the VMs with this requirement. In fact the $P$ vector includes the VM Domain-0 with the predefined value for the differentiation service fixed.

6 Future work

The proposed algorithm was implemented to adjust CPU resources, however we think that we can apply this algorithm to the network resources. This can be done following the same approach by limiting the bandwidth to a specified QoS and adjusting according to demand.

Besides using CPU intensive applications for the workload, we will setup experiments to evaluate the algorithm with a workload mix of math and web applications. We want to explore the behavior of the algorithm under different workload conditions and investigate the appropriate parameters (e.g. controller-Period) for better responses.

The differentiation service is explicitly requested by the user, but also can be mapped from the budget that an user is willing to pay or inferred from application metrics running inside the VM. In the experiments, the differentiation service remains static (we stored it in the XenStore database during VM creation) but the algorithm is able to react if this value is changed. As this value is read from the XenStore database, we just need to update it. We will study the behavior of the proposed algorithm by updating this value according to the transactions per seconds achieved by the web application. In this direction, we have a preliminary work [12] to predict the amount of CPU hertz needed to achieve or sustain a certain number of transactions executed by a given application.

References

2. VMWare Inc: Introducing vmware virtual platform, technical white paper (February 1999)


