Resource Allocation Policy for Virtualized Network Interfaces

Shashikant Gupta
Bojja Rahul Reddy

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela – 769 008, India

Resource Allocation Policy for Virtualized Network Interfaces

A THESIS SUBMITTED IN

May 2013

TO THE DEPARTMENT OF

Computer Science and Engineering

OF

National Institute of Technology Rourkela

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

The Degree of Bachelor of Technology

BY

Shashikant Gupta (Roll No. 109cs0631)

Bojja Rahul Reddy (Roll No. 109cs0654)

UNDER THE SUPERVISION OF

Prof. Pabitra Mohan Khilar

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769 008, India

Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, India. `www.nitrkl.ac.in`

Prof. Pabitra Mohan Khilar

May 13, 2013

Certificate

This is to certify that the work in the thesis entitled *Resource Allocation Policy for Virtualized Network Interfaces* by *Shashikant Gupta* and *Bojja Rahul Reddy*, bearing roll numbers 109cs0631 and 109cs0654, is a record of an original research work carried out by them under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Bachelor of Technology* in *Computer Science and Engineering*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. Pabitra Mohan Khilar

Acknowledgment

We would like to express our sincere gratitude to our Project Supervisor, Professor Pabitra Mohan Khilar, Department of Computer Science and Engineering, whose invaluable guidance and support throughout the period of this work made the successful completion of this project possible. But for his readiness for advices at all times, his educative comments and inputs, his concern and assistance even with practical things, we could not have completed our project within the stipulated period of time. We would also like to thank all professors, seniors and fellow students for their generous help and co-operation.

We extend our deepest regards and acknowledgement to all individuals whose support, motivation and encouragement in various ways provided for the successful completion of this work.

SHASHIKANT GUPTA
Roll No. 109CS0631
Dept. of CSE
NIT Rourkela

BOJJA RAHUL REDDY
Roll No. 109CS0654
Dept. of CSE
NIT Rourkela

Abstract

Over the last decade, virtualization has gained widespread importance. Virtual Machines (VMs) can now share network access in hardware, or in software or in a hybridized way. Input/Output (IO) virtualization technologies based on software utilize emulation technique, but this requires Virtualization Manager which presents central processing overhead in a significant amount. Besides, each IO operation in turn poses overhead additionally and any supported advanced capabilities inherent of physical hardware are not utilized properly. Some direct assignment based IO virtualization technologies suffer from limitations to scalability. The support for Quality of Service (QoS) may be offered within the software layers at the Virtualization Manager or Guest Operating System level which interact with the IO device that is being shared. With a preliminary investigation of the functionality of the RiceNIC (an open standard platform meant for research and education into concurrent network interface design) [5], a study of the various network interface technologies supporting IO device virtualization was carried out to precisely understand IO virtualized network interfaces. The project describes a resource allocation policy for the on-device memory of the IO device being shared, taking the instance of a complex IO device, i.e., a Network Interface Controller(NIC) supporting a reconfigurable virtualized network interface architecture design which endures multiple reconfigurable virtualized network interfaces working independently using a reconfigurable partitioned memory. It enhances the scalability of the IO device.

Keywords: Input/Output Virtualization, Virtualization Manager, Resource Allocation, Network Interface Controller, Scalability, Virtualized Network Interfaces.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is Input/Output Virtualization ?

Virtualization technologies have emerged as the key, over the last few decades, for improvements in network performance and utilization of resources. IO Virtualization increases utilization of server IO resources by consolidating more than one workloads on a single physical machine using abstraction of the underlying physical resources that are shared between multiple Virtual Machines [3].

Among the existing technologies for virtualization, the three main ones are: Full virtualization, Para-virtualization and Hardware virtualization. In full virtualization, complete hardware is abstracted virtually and operating systems, or their particular components are run without any modification, inside virtual machines. However, it involves high emulation overhead incurred by the Virtualization Manager. Then came para-virtualization in which operating system are run with modification, inside the virtual machine. In both the cases, virtualization is enabled by a software abstraction layer, the Virtual Machine Manager, that manages the virtual machines. Then came hardware virtualization with dedicated IO devices, only to have evolved now to sharing the device across multiple Virtual Machines by way of virtualized interfaces. However, for the greatest consolidation ratio and server utilization these three methods must be applied in combination [8].

IO Virtualization (IOV) exports multiple virtual views of the same physical IO device which improves scalability and performance of virtualized servers for IO device sharing by supporting sufficient number of virtual machines (VMs) necessary to use idle resources. IO virtualization requires a software for physical device management. From an adapter point of view, IO virtualization creates multiple views of the same physical device making it appear like multiple independent devices dedicated for each purpose. From a system point of view, each VM running on top of a Virtualization Manager sees its own PCI hierarchy [3].

Seeing that IEEE anticipates the need for 1 Tbps networks as soon as 2020, IOV becomes necessary because platform performance though increasing is underutilized. Hardware cost, however, is not the issue anymore, because now physical space in the data center, power/cooling costs, and management are bigger problems [1]. Besides, the ideology of go-green implies to reduce power consumption and costs. Wastage in configuring and maintaining physical components can be eliminated by IOV. Server virtualization is an efficient way of server consolidation. There are various goals of IO virtualization. To name a few, isolation; separation of memory space; almost native performance for I/O operations; separate I/O streams, interrupts, and isolation of control operations, I/O operations and errors for shared devices; scalability for sharing the IO device to optimally use idle resources [1].

The next sub-sections throw light upon the motivation for this work followed by the objective and chapter organisation.

1.2 Motivation for this work

Research in High Performance Computing (HPC) shows that almost native throughput (as in a non-virtualized environment), can be achieved by improvements in software packet handling and offloading virtualization onto the NIC [2]. So, research normally focuses on overall maximization of system throughput, rather than on the limited-resource architectures of NICs for optimal implementation of specific Quality of Service. Besides, a dynamic resource allocation policy for device partitions can thoroughly use idle memory resources and improve scalability for sharing the IO virtualized device too.

1.3 Objective of this thesis

The objective here is to devise a control strategy for memory reconfiguration at the device level to enhance scalability for sharing the IO device with inherent support for application specific requirements of the respective VMs.

We take the instance of a Network Interface Controller (NIC) - with support for a virtualized network interfaces and a reconfigurable partitioned memory organisation.

1.4 Chapter Organization

This thesis proceeds as follows: Chapter 2 gives background on related work. Chapter 3 introduces the devised policy in detail. Chapter 4 goes with the analysis and results. Chapter 5 concludes this work.

1.5 Conclusion

In this chapter, first we discussed about Input/Output Virtualization and its goals. Then we represented the Motivation for our work and Objective. Lastly we discussed about content flow of this thesis.

Chapter 2

Related work

This chapter provides background information relevant to the work in this area. An IO virtualized device is aware of the fact that it is being virtualized, so it has to implement this by presenting an interface to the Virtualization Manager that enables on-demand management of virtual devices; and to a guest OS an abstract virtual device interface permitting Virtual Machine interaction with the device, with minimum involvement of the Virtualization Manager [9]. Starting from the very beginning the various IO virtualization and sharing approaches such as software-based, direct-assignment based, Intel's Virtual Machine Device queue (VMDq) and PCI-SIG Single Root IO Virtualization (SRIOV) are discussed.

2.1 Software based IO Virtualization and Sharing Approach

Software based sharing uses the technique of emulation to provide a logical IO hardware device to the VM. The emulation layer throws itself in between the driver running in the guest OS and the underlying hardware. With this indirection, the Virtualization Manager intercepts all Input/Output traffic and interrupts generated by the driver of the guest OS. The multiple IO requests from all the virtual machines are resolved by the emulation software and serialized into a single IO stream handleable by the underlying hardware. The first common software-based IO virtualization and sharing approach is the device emulation model in which the existing drivers in the guest OS are utilized. The Virtualization Manager abstracts the HW to present each SI with its own virtual system and takes sole ownership of the underlying hardware to ensure compatibility. It intercepts and processes each IO request before passing them on to the different physical devices. Figure 2.1 represents the Software-Based IO Virtualization and Sharing Approach.

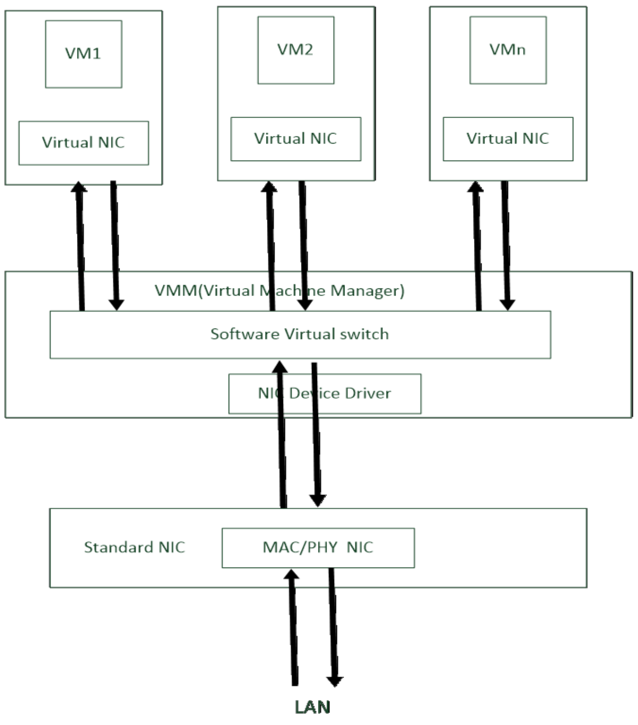


Figure 2.1: Software based IO Virtualization

A major problem faced here is that the overhead incurred by the two IO stacks traversal for each IO operation, one in the VM and one in the Virtualization Manager, is high. The second approach to software-based IO virtualization and sharing is the split-driver (para-virtualized driver) model which uses a similar approach, only that it uses a front-end driver in the guest that works in harmony with a back-end driver in the Virtualization Manager giving the benefit of no need to emulate an entire device. Drawbacks to Software-Based Sharing are that there is a significant CPU overhead incurred by the emulation layer to implement the virtual software-based packet switch and also overhead for each IO operation, which in turn reduces the throughput. Additionally, this may also eliminate the use of advanced physical device capabilities.

2.2 Direct Assignment based IO Virtualization and Sharing Approach

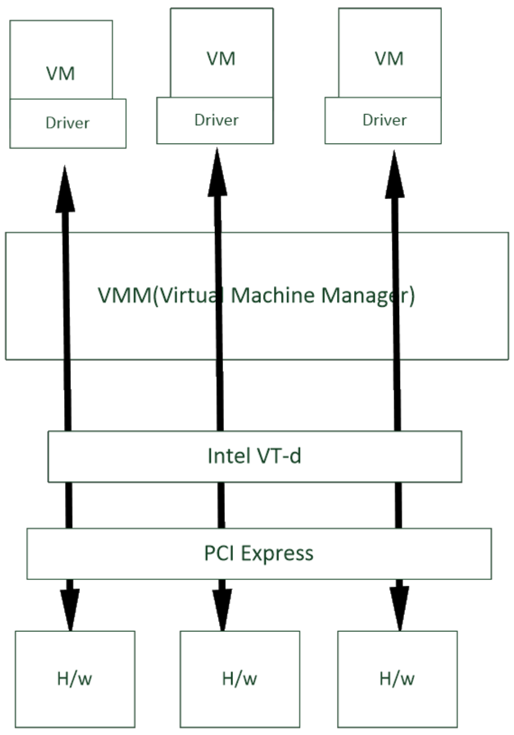


Figure 2.2: Direct Assignment based IO Virtualization

Problems with Software-based sharing can be reduced by exposing the hardware directly to the guest OS and have a native device driver up and running. The Virtualization Manager can utilize and configure an Address Translation agent (such as the Intel Virtualization Technology for Directed IO) to by-pass the Virtualization

Manager's IO emulation layer and allows a guest device driver to be able to write/read directly to/from IO device address space and the virtual machine address space. Though it provides very fast IO, the direct assignment based IO virtualization technology is nonscalable as a physical device can be assigned only to one VM. It prevents the sharing of IO devices. Figure 2.2 on the previous page represents the Direct Assignment IO Virtualization and Sharing Approach.

2.3 Virtual Machine Device queue based Approach

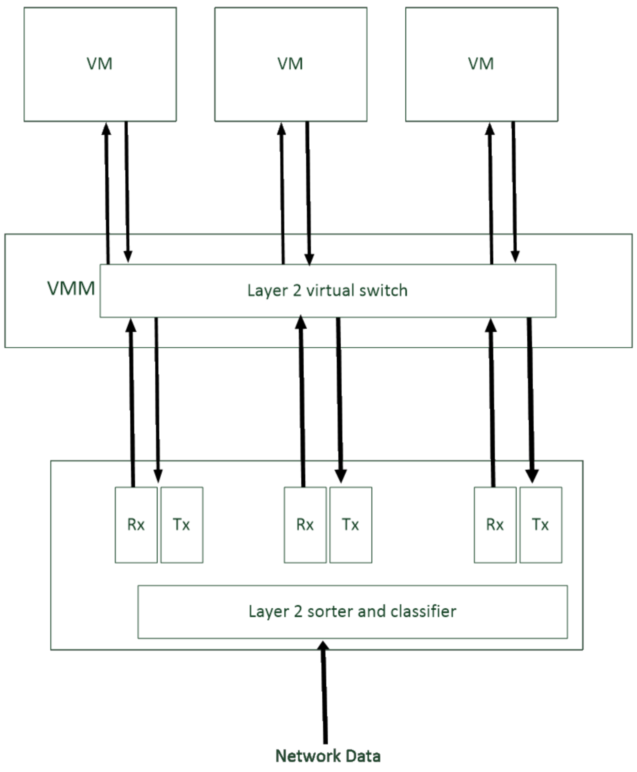


Figure 2.3: Virtual Machine Device queue based IO Virtualization

An improvement to the upper scenario is the usage of multi-queue network cards such as Intel's Virtual Machine Device queue (VMDq) network cards which offer multiple pairs of Tx/Rx queues. They create a separate queue for each VM. This allows the hardware offloading of packet (de-)multiplexing and queuing based on the MAC address (and VLAN tag) of domains [2]. The Virtualization manager assigns to each virtual machine a separate queue in the network adapter. This results in removal of overhead on the virtual switch sorting and packet routing and hence improves scalability. Besides, multiple queues remove any potential processing bottleneck by spreading the incoming load over multiple processor cores. However, the traffic still flows through the virtual switch and over normal data transports (VMBus) i.e., the VM manager and the virtual switch still have to copy the traffic from the VMDq to the VM. Figure 2.3 on the previous page represents the Virtual Machine Device queue IO Virtualization and Sharing Approach

2.4 Single Root IO Virtualization and Sharing Approach

Domains can also directly access a NIC via virtual network interfaces capable of Single Root IO Virtualization (SR-IOV). The PCI-Special Interest Group (PCI-SIG) developed the Single Root IO Virtualization and Sharing specification with the vision of overcoming the hypervisor overhead imposed on all IO operations between virtual machines and physical devices by making the device virtualization aware and providing IO device virtualization and sharing via native access to the exported multiple virtual device interfaces by means of IO page tables, virtual device identifiers and virtual device specific interrupts [3]. In SR-IOV, a physical device is mapped to a physical function (PF) which can be partitioned into multiple virtual functions (VFs) each VF still viewed as a full-edged data link. Physical resources on the NIC are then partitioned and assigned to independent virtual interfaces (vNICs). However, Quality of Service has been presumed as a software feature in the specification and is not addressed. Figure 2.4 on the next page represents the Single Root IO Virtualization and Sharing (SR-IOV).

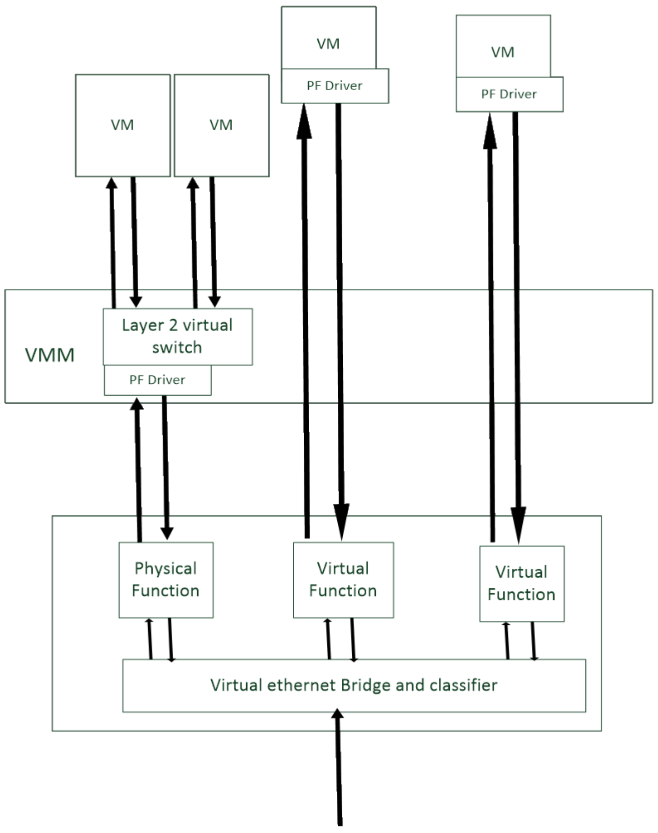


Figure 2.4: Single Root IO Virtualization and Sharing

2.5 Conclusion

In this chapter we discussed the various Input/Output Virtualization technologies. We have seen the evolution of Input/Output Virtualization from software based IO virtualization to direct assignment based approach, Intel's VMDq and PCI-Special Interest Group Single Root IO Virtualization.

Chapter 3

An Adaptive Dynamic Resource Allocation Policy

To support guaranteed application performance on virtualized servers, resource allocation mechanisms based on application's desired QoS, are needed. Also, for more flexibility these mechanisms should be closer to the resource. Ideally, these mechanisms make any unused resource available to other VM in requirement of those resources. This yields performance guarantees without losing out on resource utilization [4]. The ability to support a fair sharing of the physical NIC among the vNICs is a key requirement in network virtualization [7]. We proceed to present our approach for dynamic resource allocation in a reconfigurable virtualized network interface environment followed by the control flow and algorithms for the devised policies.

3.1 Resource Allocation Policy (RAP)

We have devised the logic for an adaptive dynamic resource allocation policy which initiated by the Virtualization Manager, continuously performs dynamic memory reconfiguration of the partitioned NIC memory among the multiple virtualized device interfaces in a fair policy-based manner, in co-ordination with the Virtualization

Manager and VMs, using the dynamic reconfiguration properties on the Network Interface Controller. The logic exists within the NIC controller. This results in better device utilization and enables the NIC to flexibly adapt to support differentiated service levels and be able to serve a scalable number of interfaces.

Depending on the type of implementation, the resource requirement information of the VM can be received from a Resource Requirement (RR) tuple, constituted of max, min and default data of each resource - a mechanism which is fairly easy to generate and widely adopted [4]. The minimum parameter provides a lower bound on the amount of memory that is allocated to the VM. The maximum parameter provides an upper bound on the amount of memory that is allocated to the VM.

The memory partition on the device acts as a dedicated buffer for each VM. At the time of virtual-NIC(vNIC) initialization, using context priority and requirements via RR, the RAP dynamically allots a default memory partition size to a VM either from the free pool or, if unavailable, through memory reclamation from lower priority VMs. The RAP consists of multiple sub-policies to handle various scenarios. Its reconfiguration policy then drives dynamic vNIC reconfiguration in terms of corresponding partition size ranging between a preconfigured maximium and mimimum configuration parameters - in accordance with corresponding VM requirements and priority and of course availability of idle memory resources.

With memory overcommitment, in which the total size configured for all running virtual machines exceeds the total amount of actual machine memory [6], we can aggressively drive up consolidation ratios. However, without proper planning and monitoring, there could be negative performance impacts resulting from excessive memory overcommitment [11]. Ballooning essentially is a cooperative operation between the guest driver and the hypervisor [10] and therefore is not used.

3.2 RAP Algorithm

The policy consists of several number of algorithms. Each algorithm of our devised

Resource Allocation Policy(RAP) are explained as follows.

3.2.1 Program Control Flow

The figure 3.1 represents the control flow of our policy. The main function first calls Single Root PCI Manager(SR-PCIM) function. It initially generates a random number of virtual machines. Then it initializes the same number of virtual NICs by using initialize function. Then main function calls dynamicallySchedule function which produces a random number and sends it to the memoryBalancer.

Then memoryBalancer performs one action among remove, add and reconfigure based on the value of random number generated by dynamicallySchedule function. The reconfigure function calls reclaimMemory function to serve the requests of higher priority vNICs. vNICadd function first calls vNICcreate function then it calls reconfigure function. If the memory is sufficient then the vNIC will be appended. If the memory is not sufficient then vNICadd function will call reclaimMemory function for the additional amount of required memory. If the reclaimMemory successfully reclaims the sufficient amount of memory from the lower priority vNICs then the new vNIC will be appended successfully otherwise the corresponding VM may be enqueued for the introduction at the next time. The figure 3.1 clearly shows all of these functions.

3.2.2 Algorithm Initialize

```
INPUT: vNICpriority[], vNICindex[]

1.   constant SCALABILITY_LIMIT : maximum limit to scalability
     array vNICpriority[] : adjacency list of vNICs with priority as header node
     array vNICindex[] : array of pointers to structure vNIC
2.   for   v = 0 : SCALABILITY_LIMIT
     do
          2.1      initialize vNICindex[v]= NULL
```

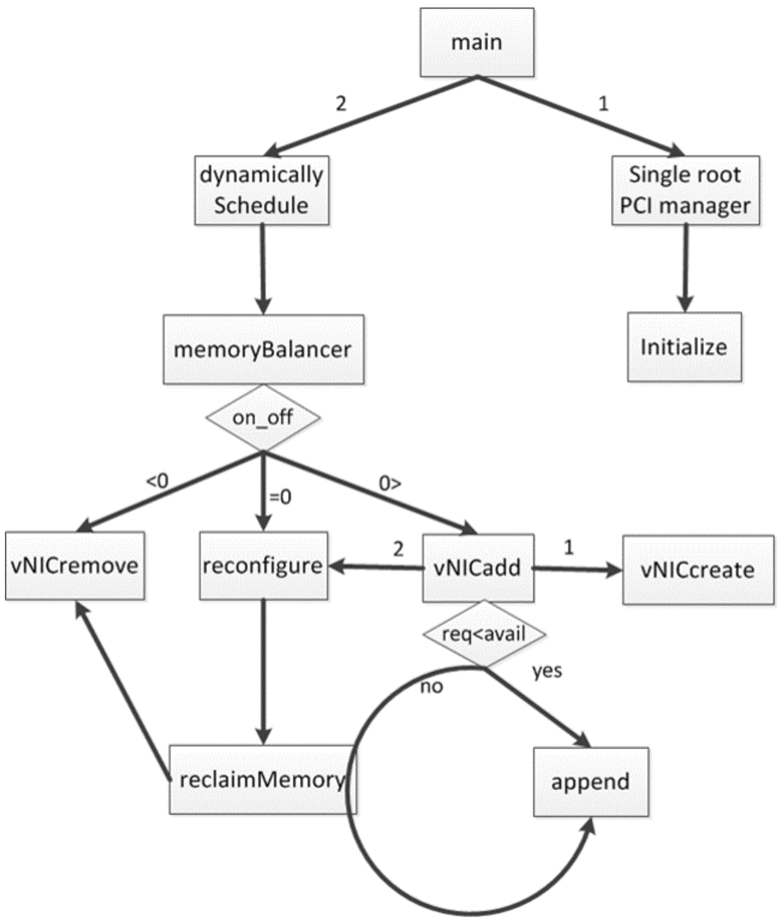


Figure 3.1: Control Flow of RAP

```
        2.2        initialize vNICpriority[v]= NULL
    done
```

Explanation

The above algorithm initializes every list of the priority based adjacency list and every index of array to NULL.

3.2.3 Algorithm dynamicSchedule

INPUT: vNICpriority[], vNICindex[], nVM, avail

```
OUTPUT: onOff (random number)

1. nVM : number of vNICs that are running at that instant.
   avail : amount of available host memory
   onOff : random number generated
2. goto step2 until the system exits
3. generate a random number ($onOff$) between $-(nVM - 1)$ $and$ $SCALABILITY\_LIMIT - nVM$
4. send $onOff$ to memoryBalancer
5. goto step1
```

Explanation

The above algorithm first generates a random number and then it passes this random number to memoryBalancer as a parameter. This algorithm runs continuously so that it looks like a real time system.

3.2.4 Algorithm memoryBalancer

```
INPUT: vNICpriority[], vNICindex[], nVM, avail , onOff

1. if onOff < 0
   then
     i. generate total onOff number of different indices which are already present
        in vNICIndex[]
    ii. for each index generated
          i. remove it from vNICIndex[],vNICPriority[]
```

```
2. else if onOff > 0
      i. add onOff number of vNICs
3. else
      i. reconfigure all vNICs according to VM requirements, priority, available
         host memory.
```

Explanation

This algorithm represents the memory Balancer logic. It takes a random number generated by dynamicallySchedule function as a input and based on that number it decides which function to call. If it is less than 0 then it removes that many number of vNICs. If it is greater than 0 then it tries to add that many number of vNICs. If it is 0 then it reconfigures all vNICs according to their priority.

3.2.5 Algorithm vNICremove

```
INPUT: vNICpriority[], vNICindex[], removeVMindex, nVM,
removeVMpriority, avail

OUTPUT: nVM(active number of vNICs)

1. Update avail
2. Determine position in adjacency list
3. Traverse the corresponding priority list to find vNIC
4. Remove vNIC from vNICpriority[]
5. Remove vNIC from vNICindex[]
6. Update number of active vNICs
```

Explanation

This algorithm helps to remove a vNIC that correponds to a dismissed VM. First it adds the partition memory to total available memory. Then it finds the vNIC to be removed in adjcency list by using its priority. Then it removes that vNIC from priority adjacency list. Then it finds vNIC position in index array and then it removes vNIC from that array.

3.2.6 Algorithm reconfigure

```
INPUT: vNICpriority[], vNICindex[], priorityLimit, nVM, avail

1. traverse vNICpriority[] till priorityLimit
2. for each vNIC in corresponding list
     i. get buffersizeReq
    ii. if (buffersizeReq <= avail)
          i. serve its request
         ii. update its new partition size and avail
        else
          i. reclaim memory from lowest priority vNIC
         ii. if reclaim failure
               i. print memory overflow and return
             else
               i. serve its request
              ii. update its new partition size and avail
```

Explanation

This algorithm implements reconfigurablity by reclaiming memory from lower priority vNICs to serve requests for higher priority vNICs. This implements the efficient use of memory.

3.2.7 Algorithm vNICadd

```
INPUT: vNICpriority[], vNICindex[], priorityLimit, nVM, avail

OUTPUT: status of new vNIC (added/enqueued)

1. create and initialize new vNIC

2. reconfigure all higher priority vNICs

3. if new vNIC partitionSize <= avail

     i. append new vNIC

   else

     i. reclaim memory from lower priority vNICs

    ii. if sufficient memory is reclaimed

          i. append new vNIC

        else

          i. print memory overflow and enque new vNIC.
```

Explanation

This algorithm adds a new vNIC. It first creates a new vNIC and assigns all of its values randomly within suitable range. Then it calls reconfigure algorithm to reconfigure all vNICs which are having higher priority than current vNIC. If the reconfigure fails then the new vNIC is enqued. If successfully reconfigured then it checks the available host memory. If it is sufficient to add new vNIC then it appends the new vNIC else it calls reclaimMemory algorithm for the reclamation of required memory. If it gives failure then this new vNIC is enqued, otherwise it is simply appended.

3.2.8 Algorithm vNICappend

```
INPUT: vNICpriority[], vNICindex[], vNICnew, nVM, avail

OUTPUT: vNICnew

1. Update avail
2. traverse vNICindex[] till first index with NULL entry
     i. add vNIC to that index
    ii. update vNIC index
3. Traverse vNICpriority[] till vNIC priority
     i. add vNIC to that index
4. Update number of active vNICs
```

Explanation

This algorithm is called by vNICadd function. It is called only when there is sufficient amount of memory to append. First it traverses to find first NULL index then it adds new vNIC to that index and updates vNIC index value. After that it reaches the header priority node based on its priority and it appends new vNIC to the end of the list.

3.2.9 Algorithm reclaimMemory

```
INPUT: vNICpriority[], vNICindex[], bufferSizeReq, currentVMpriority, nVM,
avail

OUTPUT: reclamation successful/failure

1. initialize availTemp = avail
```

```
2. traverse lower priority lists in bottom up fashion and update availTemp until
   bufferSizeReq <= availTemp
     i. if (bufferSizeReq <= availTemp)
          i. remove same number of lower priority vNICs and reclaim memory
         ii. update nVM and avail
        else
          i. return -1
```

Explanation

This algorithm first traverses the lower priority lists in bottom up fashion and updates availTemp. It continues to do this traversal until sufficient amount of memory is collected. If successul then it removes same number of vNICs and updates nVM and avail. If failure then returns -1 and the corresponing vNIC is either starved or enqued; starves in case of reconfigure and enques in case of new vNIC.

3.3 Conclusion

In this chapter, first we introduced our devised dynamic resource allocation policy. Then the control flow of this devised policy is shown. After that the devised policy is explained with the help of algorithms.

Chapter 4

Analysis And Results

In this chapter we present the analysis of the devised Resource Allocation Policy modelled in C language followed by results and conclusion.

4.1 Involved Parameters and Structures

In chapter 3 we have already seen the control flow and algorithms of the devised policy. The various parameters and data structures involved are as follows:

AVAIL_LIMIT	:	(constant)	maximum limit to the amount of available host memory
SCALABILITY_LIMIT	:	(constant)	maximum limit to the number of active VMs
RRtuple	:	(structure)	VM memory resource requirement information
VDIcontextInfo	:	(structure)	virtual context priority information
v_NIC	:	(structure)	vNIC index, VDIcontextInfo, RRtuple and partition information

noOfVM	:	(integer)	number of active VMs at any instant
freePool	:	(integer)	available host memory pool
vNICindex[]	:	(array)	array representation of pointers to vNICs in increasing order of index
vNICpriority[]	:	(adjacency list)	graphical representation of pointers to vNICs with header nodes arranged in increasing order of priority

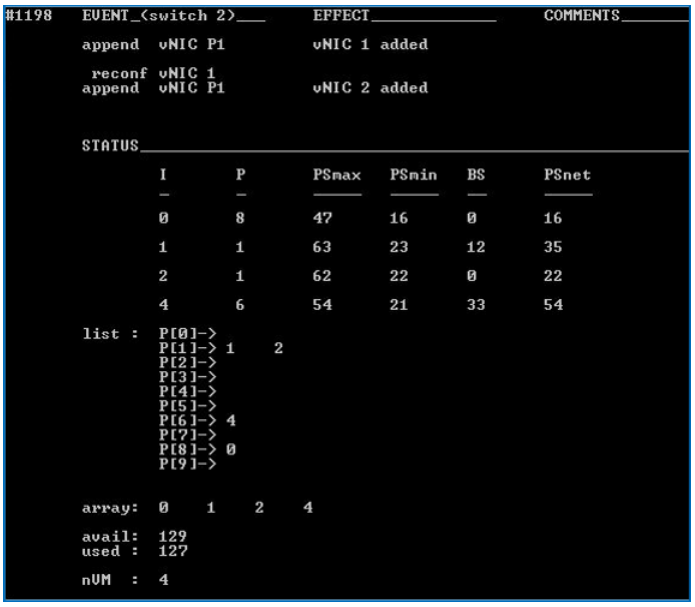
4.2 Analysis

The main module runs a number of iterations as required by using a random distribution to dynamically schedule the introduction, removal, dismissal, starvation or reconfiguration of VMs just as in any real time situation at the reconfigurable partitioned memory organization of the virtualized network interface.

For each VM the characteristics and parameters in turn, are generated too by using a random distribution while imitating practical bounds to derive maximum analogy to any scenario. The EVENT, EFFECT, COMMENTS and system STATUS are listed with each iteration.

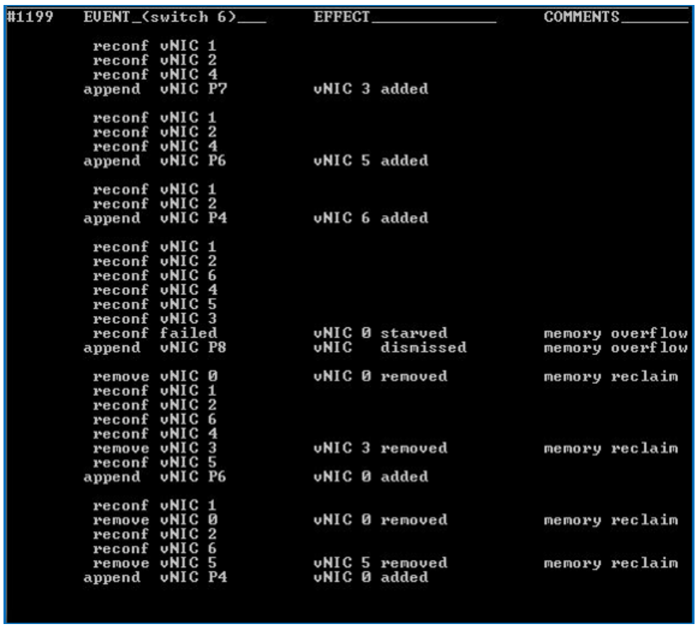
The figure 4.1 shows the 1198th iteration of RAP. In the 1198th iteration the generated random number is 2. So memory balancer checks for availability of require memory and then succesfully adds two vNICs. Observe that before adding, the reconfiguration of the already active higher priority vNICs has taken place. The total active number of vNICs after 1198th iteration is 4.

The figures 4.2 and 4.3 on the next page show the 1198th iteration of RAP. In 1199th iteration the generated random number is 6. So memory Balancer tries to add 6 number of new vNICs. Observe that to add a higher priority vNIC the policy continuously reclaims memory by removing the least priority vNICs at that instant, till required amount of memory is collected. If sufficient memory cannot be reclaimed a memory overflow message is flashed and the corresponding VM is either starved or



```
#1198   EVENT_(switch 2)___      EFFECT_____________      COMMENTS________
        append  vNIC P1          vNIC 1 added

         reconf vNIC 1
        append  vNIC P1          vNIC 2 added


        STATUS______________________________________________________________
                I       P        PSmax    PSmin    BS      PSnet
                -       -        -----    -----    --      -----
                0       8        47       16       0       16
                1       1        63       23       12      35
                2       1        62       22       0       22
                4       6        54       21       33      54
        list :  P[0]->
                P[1]-> 1    2
                P[2]->
                P[3]->
                P[4]->
                P[5]->
                P[6]-> 4
                P[7]->
                P[8]-> 0
                P[9]->

        array:  0    1    2    4

        avail:  129
        used :  127

        nVM  :  4
```

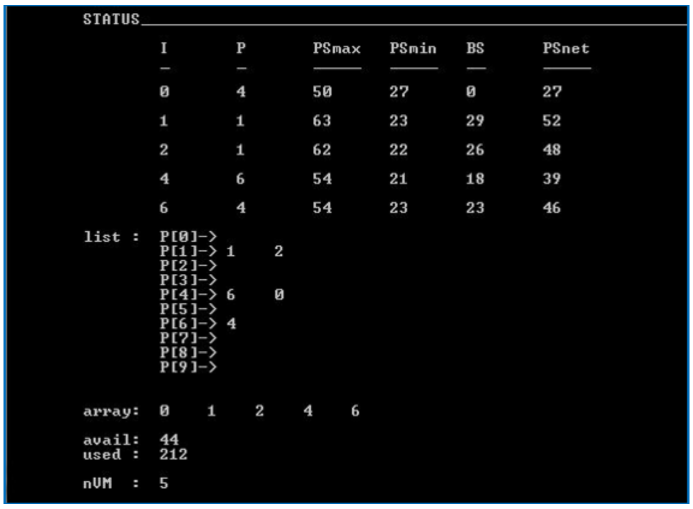
Figure 4.1: 1198th iteration of RAP



```
#1199   EVENT_(switch 6)___      EFFECT_____________      COMMENTS________
         reconf vNIC 1
         reconf vNIC 2
         reconf vNIC 4
        append  vNIC P7          vNIC 3 added

         reconf vNIC 1
         reconf vNIC 2
         reconf vNIC 4
        append  vNIC P6          vNIC 5 added

         reconf vNIC 1
         reconf vNIC 2
        append  vNIC P4          vNIC 6 added

         reconf vNIC 1
         reconf vNIC 2
         reconf vNIC 6
         reconf vNIC 4
         reconf vNIC 5
         reconf vNIC 3
         reconf failed           vNIC 0 starved           memory overflow
        append  vNIC P8          vNIC   dismissed         memory overflow

         remove vNIC 0           vNIC 0 removed           memory reclaim
         reconf vNIC 1
         reconf vNIC 2
         reconf vNIC 6
         reconf vNIC 4
         remove vNIC 3           vNIC 3 removed           memory reclaim
         reconf vNIC 5
        append  vNIC P6          vNIC 0 added

         reconf vNIC 1
         remove vNIC 0           vNIC 0 removed           memory reclaim
         reconf vNIC 2
         reconf vNIC 6
         remove vNIC 5           vNIC 5 removed           memory reclaim
        append  vNIC P4          vNIC 0 added
```

Figure 4.2: 1199th iteration of RAP

dismissed, depending on whether it is a reconfiguration or an append scenario. For every iteration EVENT, EFFECT, COMMENTS and system STATUS are updated. After the 1198th iteration, total number of active vNICs is 4 and after the 1199th



```
STATUS_____________________________________________________________
          I         P         PSmax     PSmin     BS        PSnet
          -         -         -----     -----     --        -----

          0         4         50        27        0         27
          1         1         63        23        29        52
          2         1         62        22        26        48
          4         6         54        21        18        39
          6         4         54        23        23        46

list :    P[0]->
          P[1]-> 1     2
          P[2]->
          P[3]->
          P[4]-> 6     0
          P[5]->
          P[6]-> 4
          P[7]->
          P[8]->
          P[9]->

array:    0     1     2     4     6

avail:    44
used :    212

nVM  :    5
```

Figure 4.3: 1199th iteration of RAP

iteration, though 6 new vNICs needed to be added, the total number of active vNICs is only 5. This is because the policy has added 5 new vNICs, 1 vNIC is dismissed due to memory overflow and 4 are removed in the course of reconfiguration. It can be observed that vNIC0 starved due to memory overflow.

A similar simulation was done for non-reconfigurable partitioned memory organisation to be able to compare the average number of active VMs achieved in the two cases, namely reconfigurable partitioned memory organisation and non-reconfigurable partitioned memory organisation. The results are shown in the next section.

4.3 Result

The devised policy simulation in C language for reconfigurable partitioned memory organisation was compared to a similar simulation for non-reconfigurable partitioned memory organisation to obtain the graph between scalability vs available host memory as shown in Fig. 4.4 on the next page. It clearly shows that the scalability achieved by a reconfigurable PMO is higher than that achieved by a non-reconfigurable PMO for sharing the IO device with a given amount of host

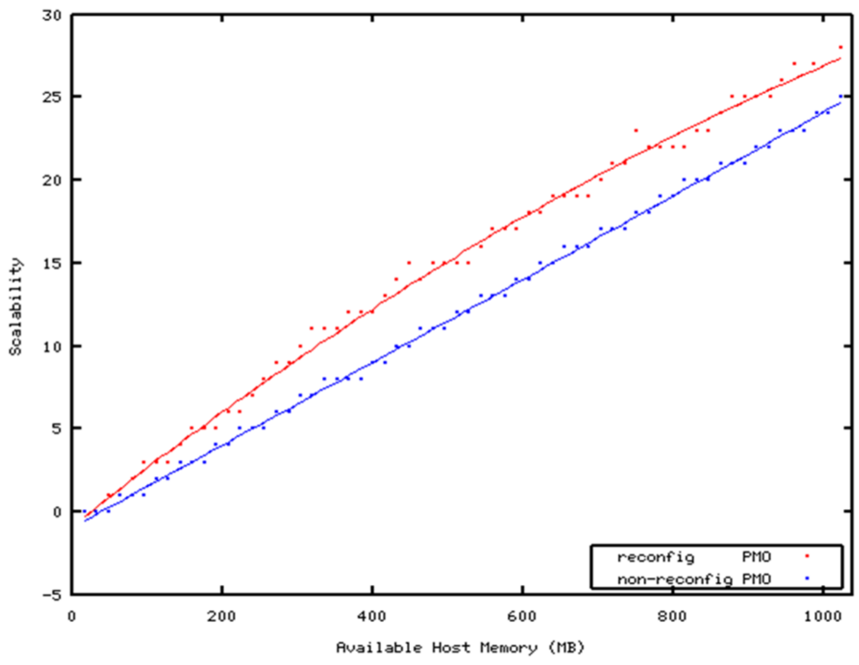


Figure 4.4: Scalability vs Available Host Memory

memory to avail.

4.4 Conclusion

In this chapter we discussed the the various sub-policies involved in RAP and presented a detailed analysis of the policy using logical modelling in C language followed by the encouraging results.

Chapter 5

Conclusions

5.1 Conclusion

As of this project with a preliminary analysis of the evolution of network interface IO virtualization we discussed a policy for dynamic resource allocation for virtualized network interfaces with a reconfigurable partitioned memory organisation. Thereafter we presented the simulation results in terms of scalability which are encouraging. Hence, we can say that the devised policy exhibits the property of scalability for I/O device sharing.

5.2 Scope for future work

This policy can be further scrutinized by resorting to analysis in more complex simulation environments.

Bibliography

[1] Patrick Kutch. Intel LAN Access Division, PCI-SIG SR-IOV Primer, Revision 2.5, January 2011.

[2] H. Rauchfuss, T. Wild, A. Herkersdorf, "A Network Interface Card Architecture for I/O Virtualization in Embedded Systems", Second Workshop on I/O Virtualization (WIOV '10), Pittsburgh, USA, March 13, 2010.

[3] PCI-SIG, "Single-Root I/O Virtualization and Sharing 1.1 Specification", January 2010.

[4] J. Lakshmi and S. K. Nandy, I/O Device Virtualization in the Multi-core era, a QoS perspective, in the Proceedings of the 4th International Conference on Grids and Pervasive Computing, as part of the 1st International Workshop on Grids, Clouds and Virtualization, Geneva, Switzerland, May 4-8, 2009.

[5] J. Shafer, S. Rixner, RiceNIC: A Reconfigurable Network Interface for Experimental Research and Education, Workshop on Experimental Computer Science, San Diego, CA, June 2007.

[6] Carl A. Waldspurger, "Memory Resource Management in VMware ESX Server", Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Boston, Massachusetts, December 9-11, 2002.

[7] Sunay Tripathi, Nicolas Droux, Thirumalai Srinivasan, and Kais Belgaied, "Crossbow: from hardware virtualized nics to virtualized networks," in

Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures, New York, NY, USA, 2009, VISA '09, pp. 53-62, ACM.

[8] VMware, Inc. white paper. Virtualization: Architectural Considerations and Other Evaluation Criteria, Nov 9, 2005.

[9] Himanshu Raj, Karsten Schwan (2005) Implementing a Scalable Self-Virtualizing Network Interface on an Embedded Multicore Platform, in Proceedings of WIOSCA 2005, in conjunction with IISWC 2005, Austin, TX.

[10] Chin-Hung Li, "Evaluating the Effectiveness of Memory Overcommit Techniques on KVM-based Hosting Platform," World Academy of Science, Engineering and Technology International Conference Program, October 24-25, 2012.

[11] Kingston Technology white paper. The Yin and Yang of Memory Overcommitment in Virtualization : The VMware vSphere 4.0 Edition, 2010.