Wireless Pers Commun (2013) 73:1513–1528
DOI 10.1007/s11277-013-1263-0

# VRAS: A Lightweight Local Resource Allocation System for Virtual Machine Monitor

**Hai Jin · Wei Gao · Song Wu · Xuanhua Shi**

Published online: 11 June 2013
© Springer Science+Business Media New York 2013

**Abstract** Traditional computing resource allocations in virtualization environment devote to provide fairness of resource distribution when the overall workload of host is heavy. That makes those allocations lack of efficiency under light workloads. To target this, we design and implement a lightweight resource allocation system, virtual resource allocation system (VRAS). Considering the fact that workloads can be balanced by migrating virtual machines to other hosts, we propose a request driven mechanism to focus on resource allocation under light workloads. We also present some allocation strategies used in VRAS to explain how it works on processor and memory resources. Our experiment results demonstrate that VRAS can result in throughput improvements of 28 % for RUBiS application, and the network overhead reduction of 81 %, comparing with the traditional allocation methods.

**Keywords** Virtualization · Server consolidation · Resource allocation · Request driven mechanism · Workload

## 1 Introduction

One feature of the virtualization technology that makes it an important cloud enabling technology is the encapsulation of the computing environment [1]. The virtual machine monitor (VMM) provides dedicated processor, storage and network devices to each virtual machine (VM), and the operation system working inside the VM, called guest OS, controls these virtualized resources independently. For raising resource utilization and saving overall cost, server consolidation could be achieved by simultaneously running many VMs on one physical machine [2]. Because the server's performance is subjected to the restriction of available resources, each VM is usually over-provisioned to meet vast service requests. However, the

H. Jin (✉) · W. Gao · S. Wu · X. Shi
Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
e-mail: hjin@mail.hust.edu.cn

S. Wu
e-mail: wusong@hust.edu.cn

demands of a service are not always reach the upper limits, but often varying with time lasting. As a result, a VM would keep under-utilized for a long time, and much of the occupied resources are wasted while other VMs can not consume them. That situation also restricts the number of consolidated servers. Allocating the VM's virtualized resources to a reasonable range can reduce considerable waste at most time, but lead VM to be over-loaded sometimes. Unfortunately, on many virtualized data centers, servers running such as web applications and databases are functionally connected while they exist in isolated VMs [3]. The overall performance of services will be undermined by even one congested VM [4]. Therefore a mechanism that allocates proper resource to the VMs based on their respective instant requirements in time is needed to be imported into the virtualization infrastructure. Such mechanism shares certain part of resources as *flexible resource* among VMs. Usually those redistributable resources are not frequently used by one VM.

On a data center having sufficient resources, the object of that kind of resource management mechanisms is to ensure that no service will meet its performance bottleneck due to the lack of CPU execution time, memory size, network bandwidth or something else [5], and try raising the resource utility rate as well as possible at the same time.

Traditional researches put much works on how to quickly and fairly allocate resources to the VMs on heavy loaded hosts. However, due to the VM migration technology [6,7], VMs can be moved among a group of connected hosts to find proper one that meets their resource requirement. If a physical machine lacks of resources, its overloaded VMs could be migrated to another host with sufficient resources. So we argue that in a practical virtualization environment, hosts could have enough resources to run VMs on them at most time, and resource scheduler should take more considerations to allocate resource in host under light or moderate workload.

We in this paper propose a virtual resource allocation system (VRAS) to manage all VM's resource on a physical host. The main goal of VRAS is to dynamically share the computing resources across VMs without too much extra cost introduced to the system. So we suggest that our solution should be quite simple and efficiently perform allocation with less overhead under the condition that host has enough resources. In VRAS framework, each VM detects its resources usage and sends request signals to the VRAS manager running on physical host. The resource allocation manager will re-calculate the VM's resource quota based on allocation strategy, and then add or remove the VM's resource.

The rest of this paper is organized as follows. Section 2 presents related work on virtual resource allocation. Section 3 introduces framework of the VRAS. Section 4 presents the details of VRAS implementation on processor allocation. Section 5 describes VRAS's memory scheduling mechanism. Section 6 evaluates the system's performance towards CPU and memory resources through experiments on different benchmarks. At last, we conclude the paper and discuss the future development of the system in Sect. 7.

## 2 Related Work

Resource management [8,9] has been studied to allocate different types of resources towards specific applications such as web server [10–12]. They often require the application to be modified, thus are not suitable to apply to the virtual machines directly.

In virtualized environment, resources used to be statically allocated to every VM [13,14]. Many previous works such as allocation for overloaded VM [15], nonlinear adaptive control [16], and others [17,18] have been proposed to study dynamically managing resources. They basically pay attentions to accurately allocate virtualized resources under different conditions,

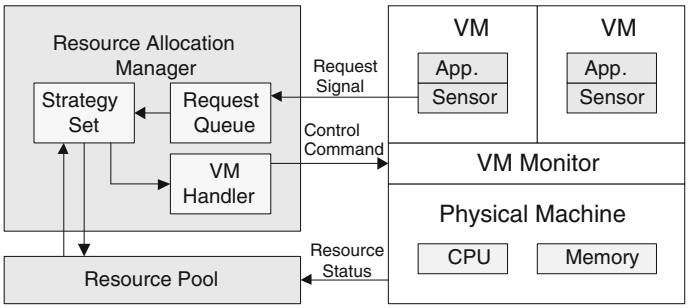


**Fig. 1** VRAS architecture

but do not consider how to perform the job with agility. Some of them are too complicated that could lead to terrible overhead under certain circumstances.

In order to effectively control the VM resource, FVM [19] is proposed as a self-adaptive mechanism, which makes VM actively adapt the host's resource distribution. It emphasizes the fairness in VM's resource allocation, and works well when the whole system are load saturated, but it needs VMs themselves limiting their resource requirements to adapt resource conflicts, resulting in the loss of application's QoS when too many VMs are consolidated.

Other approaches try to avoid resource absence on a host. Ghost VM [20] prepares many standby VMs on every host. When the service demand is increased, it activates one more VM on a light load host. Sandpiper [21] just migrates the overloaded VM to a new physical machine to balance the overall workload.

The main feature of our work differs from previous research efforts lies on the fact that the request driven mechanism reduces the communications between VM and allocator. Based on that, we employ simple strategy to allocate resources effectively in the light workload environment, avoiding the fairness consideration under heavy workloads.

## 3 VRAS Framework

VRAS is a general resource management framework for VMs running on the same host. Instead of those using complicated approaches to deal with the specific circumstances, VRAS allocates different types of resources based on a request driven mechanism. As Fig. 1 shows, VRAS maintains a resource pool to record the status of total available resources and resources assigned to VMs on a physical machine. Resource allocation manager (RAM) is the main part of VRAS framework which performs the most of the allocation tasks. Sensor in each VM collects resource usages and requirements and sends them to RAM.

There are three steps to dynamically allocate resources to VM:

1. Resource monitoring: Sensor in a VM gathers information about how each VM uses the current assigned resources, predicts its resource requirement of next phase, and then sends the request signals;
2. Decision making: when a VM's resource requirement is changed, RAM calculates the new quota to it based on available resource in Resource Pool and proper allocation strategy;
3. Resource allocating: RAM sends control commands to adjust the VM's resource assignment and updates the overall resource distribution information in Resource Pool.

Sensor running in a VM is employed to detect the resource status, such as CPU usage and memory usage. The prediction of resource requirement in our mechanism is subject to the

type of the resource and its allocation strategy. The accuracy of prediction often relies on the cost of computing resources. In this paper, we simply use the current CPU usage as the CPU requirement and current used memory size as the memory requirement in the next phase. To reduce the system resource spent on communication between Sensor and RAM, those status data need not to be sent at every time. Sensor only reports the resource's new requirement when the usage of one resource gets changed to a curtain threshold value. VRAS is designed to support configurable resource allocation strategies. For each strategy, the sensor collects VM's specific running status by invoking proper API of a guest OS, the frequency of collection and the method of prediction are decided by the strategy as well. Sensor communicates RAM with regular request signal. There are two types of request signals sent by Sensor for each resource: *Apply* and *Return*. When one resource's usage raises over the upper threshold, Sensor will send *Apply* signal to RAM alone with the resource status of VM to acquire more resource; and *Return* signal will be sent when the resource's usage is less than the lower threshold, then the free resource will be taken back to the resource pool of the host.

RAM performs the resource scheduling according to the preloaded strategies. It puts all received request signals into a Request Queue, and then passes them to the Strategy Set one by one to calculate new resource quota of the VM. At last, the VM Handler controls the VM Monitor to complete the allocation process. A request includes information as VM's domain id, timestamp, resource type (processor or memory), request type (apply or return resource) and other details of resource requirement. When a new request arrives, it will be added to the tail of Request Queue and then wait to be answered. In case that a request in the queue have not been scheduled while another request comes from the same VM applying the same resource, which makes early scheduling unnecessary, RAM compares the arrived request with the requests in the queue, and replace the request having same domain id and resource type in the Request Queue with the newest one. Each time RAM picks the request from the head of the Request Queue, updates the request's state to *processing* and starts the resource allocation task. RAM provides a Strategy Set to meet different situations in resource allocation tasks. Based on the resource type and request type, the proper strategy will be chosen, and relative parameters read from the request will be sent to it. The strategies in the set are designed to be simple and efficient, and easy to be reconfigured, even can be replaced. We discuss the allocation strategies in next two sections. Generally, the strategy analyzes the resource requirement, checks the free resources in pool, and decides how many resources would be given to or taken back from the target VM, then sends instructions to adjust VM's resource allocation and updates the Resource Pool. The VM's resource configuration is managed by the VM Monitor. RAM uses VM Handler to generate the control commands according to the result output by strategy and communicate the VM Monitor to perform reconfiguration of the VM's resource.

The Resource Pool maintains the resource distribution status of the physical machine, including resources allocated to every VM and total free resources that have not been assigned to any VM. Those data are continuously acquired from the VM Monitor. When RAM reconfigures a VM, the modification will be immediately updated to the Resource Pool and kept for a short time to avoid the resource been reallocated by other requests before the current operation is finished (succeeded or failed). After that reserve period, the status will be refreshed with new data from the VM Monitor.

VRAS is designed to reduce the communication cost in the framework, especially between RAM and VM. By introducing request driven mechanism, Sensor only sends request signals when the resource usage of the VM is out of the acceptable range. RAM need not repeatedly acquire information from the VMs.

## 4 Processor Resource Allocation

The virtualized processor (VCPU) in VM can be considered as a process running on a physical CPU. Virtualization technologies such as Xen [13] provide CPU schedulers [22] to manage sharing physical CPU resources among the process-like VCPUs. Those schedulers allocate processor resource by assigning different CPU time slices to execute each VCPU in a period, and perform effective and fair distribution of the CPU time slices. Based on one most popular scheduler used in Xen, the Credit Scheduler [23], two types of strategies can be introduced to dynamically allocate processor resource to meet the application's real-time requirement:

Fine-grain allocation adjusts each VM's CPU quota to control how many CPU time slices can be assigned to the VCPUs in a period based on the running status of VMs. Some VMs often try to occupy too much CPU time when they are processing a large task or being handled, such as being deployed or being migrated. Those unstable states would make overall resource insufficiency and impact other VM's performance as a result. The fine-grain allocation is designed to restrict the CPU slices that a rough VM can acquire based on the available resources of the physical machine, so that the processor resources of other VMs can be guaranteed.

Coarse-grain allocation controls each VM's available VCPUs by reconfiguring the VCPU number of the VM to meet its resource requirement. Obviously, though a VM's VCPUs can be set as many as its guest OS can support, that number generally should not exceed the amount of the host's physical CPUs. Furthermore, the solution that assigns maximum number of VCPUs to all VMs will makes unnecessary scheduling cost wasted on void VCPUs. So, the principle of the coarse-grain allocation is to provide enough VCPU to every VM based on the CPU usage and take back any VM's void VCPUs, except it is the last one of the VM.

To perform those two types of processor resource allocations, a simple strategy is proposed in VRAS, shown in Algorithm 1.

Before the allocation, Sensor in the VM collects running states of VCPUs in real time usage and sends request to RAM if the average VCPU usage breaks the upper or lower threshold. For instance, if the current VCPU usage is more than 75 %, Sensor will send *Apply* CPU signal to RAM; otherwise if the current VCPU usage is less than 30 %, Sensor will detect if there is any VCPU in a blocked (waiting for resources to execute tasks) or offline (no task assigned) state, then send *Return* CPU signal if there are free VCPUs in the VM.

The beginning of the strategy is the preparation part (Line 1–4), including locating the target VM given by the domain ID in the request, reading the VCPU weight that set to the VM for its normal running, current VCPU number assigned to the VM and total available CPU number of the host physical machine. Here the weight of the VM's VCPU represents the share of CPU time slice that the VM can utilize comparing to other VMs on the same host.

For the *Apply* CPU request, the strategy conducts coarse-grain allocation (Line 6–8) by adding one more VCPU to the target VM. Xen supports hot-plug VCPU to its VM, so the VM can immediately utilize more processor resource from added VCPU. In that process, the VCPU number of the VM is always constrained within the host's maximum number of physical CPU. If a full VCPU configured VM still sends request to apply more CPU resource, it means the overall CPU resource of the host is going to be insufficient, then RAM will report warning to the system manager (Line 9–11). When the target VM gains more than half number of total processors, the fine-grain allocation (Line 12–14) will be activated. The VM's VCPU scheduling weight will be adjusted down to a certain percentage (here we set 50 %) of its normal value, to prevent over-occupancy of processor resource coming from that VM.

The processor resource assigned to the VM can be retrieved in the same way. As the response to the *Return* CPU request, the strategy reduces the target VM's VCPU number by

one in allocation each time, unless the VM has only one VCPU configured (Line 16–19). The VCPU scheduling weight will be restored to the normal value when the VM's VCPUs are released down to half of the host machine's processor number (Line 20–22).

Algorithm 1: AllocateCPU( *R* )

```
1:  VM ← get the VM according to R's domain ID
2:  W ← VM's normal VCPU weight
3:  N ← VM's current VCPU number
4:  M ← Available physical CPU number in resource pool
5:  IF R's request type is "apply" THEN
6:    IF N<M THEN
7:      N ← N+1
8:      set VM's current VCPU number to N
9:    ELSE
10:     send CPU resource insufficiency message
11:   END IF
12:   IF N>M/2 THEN
13:     set VM's current VCPU weight to W/2
14:   END IF
15: ELSE IF R's request type is "return" THEN
16:   IF N > 1 THEN
17:     N ← N-1
18:     set VM's current VCPU number to N
19:   END IF
20:   IF N<M/2 THEN
21:     set VM's current VCPU weight to W
22:   END IF
23: END IF
```

On a light loaded host, the control of each VM's VCPU number determines the processor resource's distribution. When CPU resource contest occurs on a heavy loaded host, the adjustment of each VM's weight takes effect. The combination of those two strategies promises the accuracy and fairness of processor resource allocation in both light and heavy workload conditions.

## 5 Memory Resource Allocation

Popular virtual machines use a *ballooning* technique [24] to support changing a VM's available memory spaces during its running time. The *ballooning* technique can steal free mem-

ory from a VM and the ballooned-out memory can be reassigned to another VM, without redeploying or restarting of the target VM required. That makes dynamic memory resource allocation possible, and leaves the allocation strategy two questions: (1) when adjust the VM's available memory size and (2) how many memory pages should be given to or taken away from the VM. The first question implies that the allocation should be performed as soon as possible when the VM's memory requirement changes. In our VRAS framework, Sensor can detect the variation of memory usage at first time and immediately decide which response is expected to deal with that variation. However, the second question will not be solved as easily as the former one. Due to the isolation of VM's memory, a memory page can be used by only one VM in a moment. Like CPU time slices, memory pages are the smallest resource unit allocated. Once a page is written by a VM, it cannot be accessed by another VM unless it is released. Unfortunately, there is not a fine-grain approach that can frequently and precisely schedule memory pages like credit scheduler to CPU time slices. Therefore the purpose of typical *ballooning* based memory resource allocation approaches is to provide not only VM's current needed memory, but also extra memory space for meeting possible increasing of memory requirement in the following period before the next round of allocation.

VRAS allocates memory resource on the basic concept of the VM's real memory usage, which is defined as:

$$U = \frac{M_c - M_b}{M_{\max} - M_b} \tag{1}$$

where $M_c$ is the size of current used memory, $M_{max}$ is the maximum size of memory configured to the VM when it is deployed, $M_b$ is the size of balloon memory returned to the host's memory pool. Because the balloon memory is recognized as used in the target VM, the memory usage $U$ is the ratio indicates actually occupied share of all allocated memory in the VM. Also, $U_{min}$ and $U_{max}$ are introduced as threshold of normal memory usage. $U$ will be considered *healthy* if the value is between $U_{min}$ and $U_{max}$, otherwise Sensor will send *return* or *apply* signal to RAM.

By resizing the balloon memory, the memory allocation strategy tries to make the real memory usage to the expected ratio. Assuming the VM's used memory size $(M_c - M_b)$ will not change, thus for achieving the memory usage to be $U_1$, the balloon memory needs to be adjusted to:

$$M_{b1} = M_{\max} - \frac{U_0}{U_1}(M_{\max} - M_{b0}) \tag{2}$$

where $U_0$ and $M_{b0}$ represent the memory usage and balloon memory before the allocation, respectively. Since $(M_{max} - M_b)$ means total allocated memory to the VM, the allocation process can be simplified to adjust the VM's available memory size to:

$$M_{a1} = \frac{U_0}{U_1} M_{a0} \tag{3}$$

where $M_{a0}$ is the memory size allocated before current round.

The whole memory resource allocation strategy is explained by Algorithm 2.

The strategy firstly reads the current allocated memory and its usage from the located target VM (Line 1–3). The size of total free memory that can be allocated is also read from the resource pool (Line 4). Then according to the Eq. (3), a new memory size is calculated for the allocation (Line 5), here we set the objective memory usage to 0.6 (60 %). This new memory size can be set to the VM for both *Apply* memory and *Return* memory allocation.

The new memory size could be amended when it is used to add more memory to the VM. The strategy needs to make sure that the allocating memory does not exceed the maximum

size configured to the VM (Line 7–9), and the added part does not exceed the available memory in the resource pool neither (Line 10–15). The message declares "out of memory resource pool" will be sent when all the free memory in the pool is allocated to VMs. When the allocating memory size is revised, the allocation is finished after the new size is set to the target VM, and the memory pool makes the corresponding change (Line 16–17).

As for the allocation that returns VM's void memory to the resource pool, each VM is assigned a property expresses the minimum memory size for its essential running. The strategy checks the allocated memory size and makes it not less than that reserved size (Line 19–21). Then the VM and the resource pool are handled at the end of allocation (Line 22–24).

Algorithm 2: AllocateMemory(R)

```
1:  VM ← get the VM according to R's domain ID
2:  A0 ← VM's current allocated memory size
3:  U0 ← VM's current memory usage
4:  M ← Available free memory size in resource pool
5:  A1← U0*A0/0.6
6:  IF R's request type is "apply" and A1 > A0 THEN
7:      IF A1 > Mmax THEN
8:          A1 ← Mmax
9:      END IF
10:     D ← A1 – A0
11:     IF D > M THEN
12:         D ← M
13:         A1 ← A0 + M
14:         send memory resource insufficiency message
15:     END IF
16:     set VM's current allocated memory size  to A1
17:     set free memory size in resource pool to (M  – D)
18: ELSE IF R's request type is "return" and A1 < A0 THEN
19:     IF A1 < Mmin THEN
20:         A1 ← Mmin
21:     END IF
22:     D ← A0 – A1
23:     set VM's current allocated memory size  to A1
24:     set free memory size in resource pool to (M  + D)
25: END IF
```

The primary consideration of memory allocation strategy in VRAS is to provide VMs on the host enough memory to guarantee the QoS of applications running on them. Once the

**Table 1** VRAS configuration

| Parameter | Description | Value |
|---|---|---|
| pu_min | Lower threshold of VCPU usage | 30 % |
| pu_max | Uppper threshold of VCPU usage | 70 % |
| mu_min | Lower threshold of memory usage | 40 % |
| mu_max | Upper threshold of memory usage | 80 % |
| ms_min | Size of VM's reserved memory | 128 MB |
| ms_max | Size of VM's configured memory | 2 GB |

memory is allocated to a VM, it will not be taken back to resource pool until the VM send *Return* memory request. Thus no *Apply* memory request will bring memory supplies to the VM when the memory resource pool is empty. In that case, raising memory usage's lower threshold and allocation objective will help moving more void memory from light loaded VMs to heavy loaded VMs, which improves the fairness of memory allocation.

## 6 Evaluation

In this section, we present experiments conducted to evaluate the performance of VRAS. The benchmarks used in experiments include a computing-intensive program, a memory-consuming application and a Web application. The experimental results demonstrate that VRAS effectively allocates both processor and memory resources with tiny communication cost.

### 6.1 Experiment Setup

All the experiments are conducted on a two-socket server; each socket has 4 Intel Xeon 1.6 GHz CPUs. The server is equipped with 4 GB DDR RAM, 160 GB storage, and 1,000 Mbps Ethernet network connected. Linux 2.6.18 with Xen 3.1.0 installed as the host operation system. Every VM deployed for experiments uses RedHat Enterprise Linux 5 as guest OS. VRAS used in the experiments is configured as Table 1:

To compare with the typical resource allocation mechanism, we conduct experiments on VNIX [25], which continuously collects every VM's running status and uses additive-increase/multiplicative-decrease (AIMD) strategy [9] to allocate processor and memory resources. We also statically configure the VM with full resources as the best case, in which the VM always runs with most resources that can be assigned and the resource reallocation will not happen.

### 6.2 Computing-Intensive Application

To evaluate the effectiveness of processor resource allocation in VRAS, we conduct a set of experiments using Super PI [26]. Super PI is a single thread computing-intensive program. We execute 1–8 individual Super PI jobs simultaneously in a VM, each job calculates pi to 1 million digits after the decimal point. We also run jobs in the VM statically assigned with 8 VCPUs and 2 GB memory for the best case. Each time before the Super PI jobs start running, the VM would clear previous jobs and wait for a short time, so that the allocation systems can withdraw VCPUs and leave just 1 VCPU for the next test.

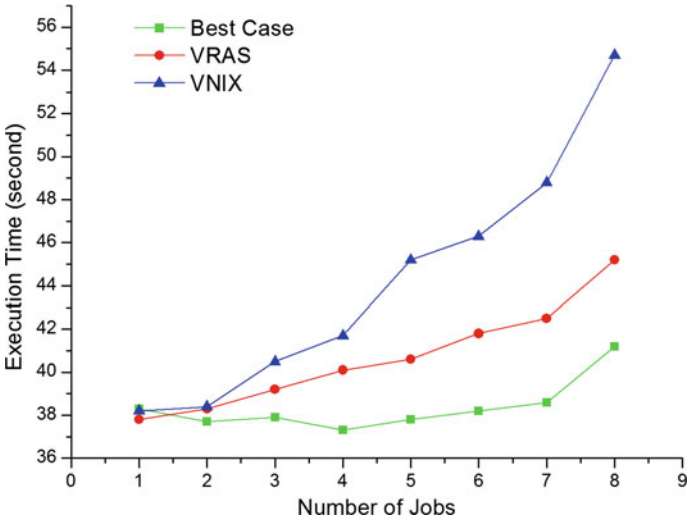


**Fig. 2** Super PI performance

From Fig. 2 we can observe that, comparing to the best case, both dynamic allocation systems will take time to add VCPU to the VM as the number of jobs increasing, which makes loss of efficiency. Considering the experiments bring most tough requirements to the processor resource, those declines would be less in usual cases.

As the number of jobs increases, VRAS performs better than VNIX. The request driven mechanism of VRAS calls for allocation immediately when resource requirement changes. While in VNIX the interval between two rounds of VM status collection may lead to delay of allocation. The results show that VRAS improves the efficiency of the 8-jobs task by 21 % comparing to VNIX.

### 6.3 Memory-Consuming Application

This part evaluates the effectiveness of memory resource allocation in VRAS. We transfer a group of files from the host to the local VM through SCP connection in the experiments. The files are sized from 128 to 1,024 MB. Despite the disk I/O and network I/O, the performance of big file transfer is sensitive to the memory size. Each file is respectively transferred with VRAS, VNIX and full resource configured VM as best case. At the beginning of every experiment, the VM's resource will be set back to the initial status (a single VCPU and 128 MB memory) when using dynamic allocation systems.

As shown in Fig. 3, VRAS does not lose too much performance comparing to the best case, which suggests that it allocates required memory to the VM efficiently. However, due to the AIMD strategy, VNIX shows degradation on memory allocation when resource requirement increases. The advantage of VRAS reaches 69 % of performance improvement to VNIX when transferring 1,024 MB file.

### 6.4 Web Application

In this part we evaluate the general performance of VRAS using RUBiS [27], an auction site benchmark. Three VMs are deployed for running a database system, a web application

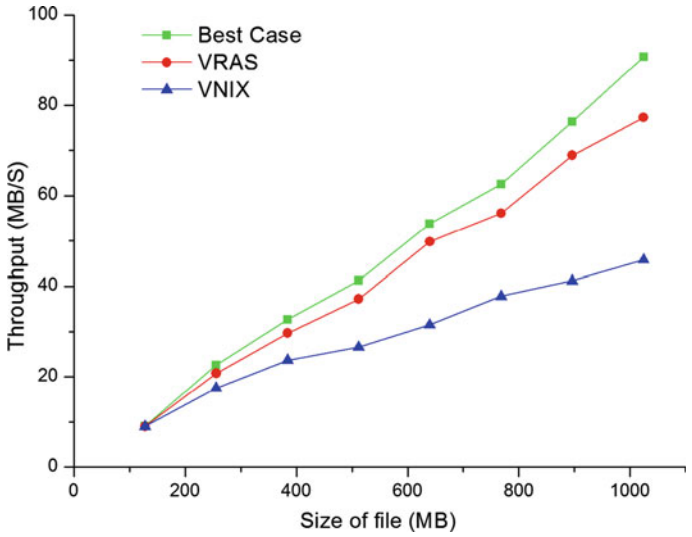


**Fig. 3** File transfer performance

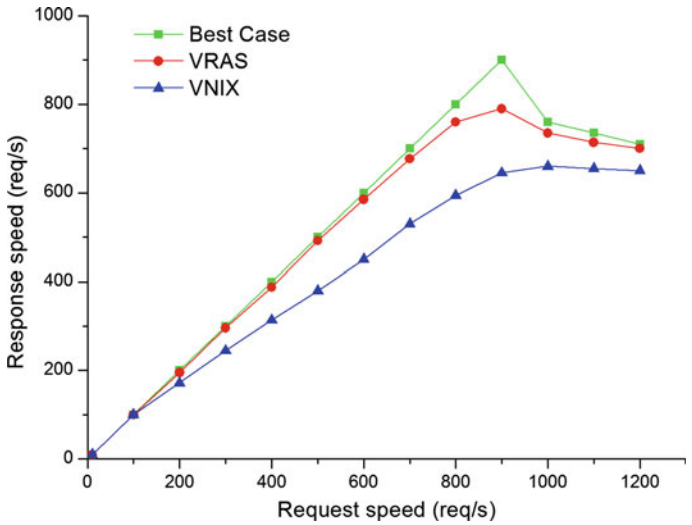


**Fig. 4** RUBiS performance

server and a client, respectively. The experiments are conducted on VRAS, VNIX and best case like before, only in the best case configured 1.6 GB memory to the web server VM and database VM, 512 MB memory to the client VM. We use the client to send requests to access the RUBiS web site in different speeds, and count how fast they respond. The results are shown in Fig. 4.

The results demonstrate that VRAS generally performs better than VNIX, the former behaves 28 % better than the latter at the speed of 800 req/s. In conditions of light workloads (request speed is less than 600 req/s), the efficiency of VRAS is very close to the best case.

QoS of application is not the only concern of a resource allocator, another important purpose of resource dynamic scheduling is to squeeze void computing resources from the

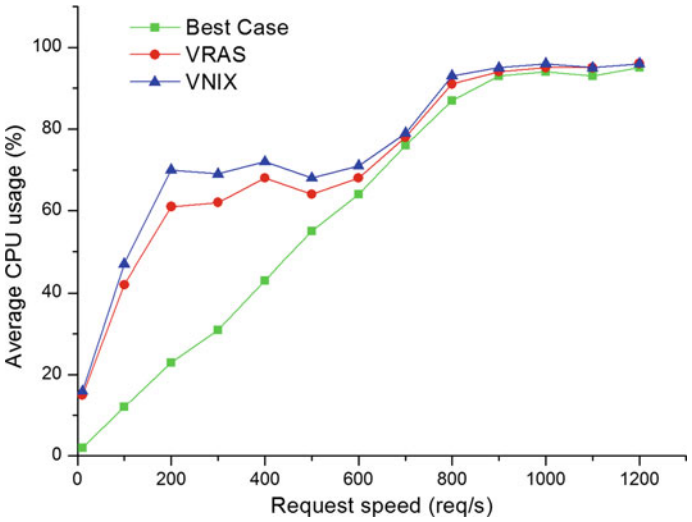


**Fig. 5** RUBiS web server's CPU usage

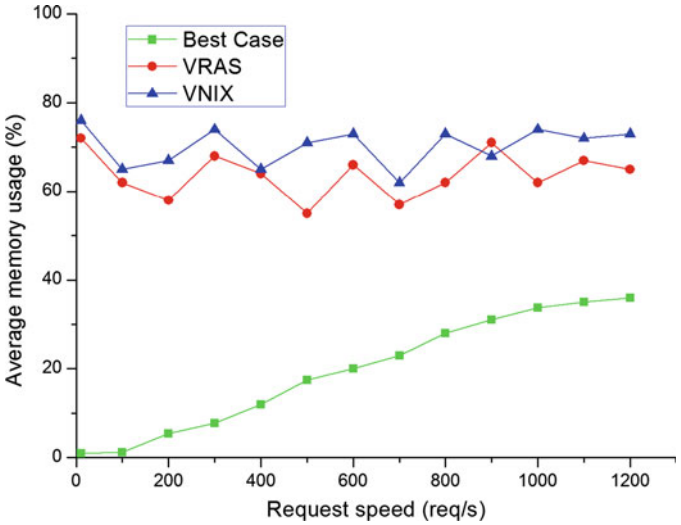


**Fig. 6** RUBiS database's memory usage

VM and raise the VM's resource usage. Figure 5 shows the average CPU usages of VM in the RUBiS experiments, which are calculated by the following equation:

$$U = \frac{1}{T}\sum_{t=1}^{T}\left(\frac{1}{n_t}\sum_{i=1}^{n_t} u_{ti}\right) \tag{4}$$

where $T$ is the time spent for the experiment, $n_t$ is the VCPU number of VM at time $t$, $u_{ti}$ is the usage of the $i$th VCPU at time $t$. We observe that both VRAS and VNIX keep CPU usages between 60 and 70 % when the workloads are light. Alone with the raising workloads,

VRAS allocates full processor resources to the VM as well as VNIX, and makes the CPU usages very close to the usages in best case. The improvement also happens to the memory resource. As shown in Fig. 6, the average memory usages are at the position around 65 % in VRAS and VNIX, while climbing from 1 to 36 % in best case (2 GB memory configuration). That is, more than 1 GB of the memory is saved from wasting in the VM which does not need so much resource. We also notice that the average memory usages of VNIX are higher than that of VRAS in most cases. Considering the difference of performance impacts to the application in VM caused by those two allocators, it implies that VNIX schedules resource not as quickly as VRAS does.

### 6.5 Overhead Analysis

Overhead of VRAS mainly comes from two places: RAM in host and Sensor in VM. The cost of Sensor is relatively stable, while that of RAM depends on the number of managed VMs and the variation of each VM's resource requirements. The time complexity of algorithms used in VRAS resource allocation strategies is $O(n)$, here $n$ denotes the number of received requests. That value could be increased if other complex strategies, such as prediction algorithms on resource consumption, are employed.

One contribution of VRAS is the low overhead on network bandwidth. During the experiments, we observe that the average throughput of VRAS reaches no more than 0.3 kbps. It is much less than 1.6 kbps of VNIX, reduces 81 % of the network bandwidth consumption. Theoretically, even in the toughest situation, the request driven mechanism used in VRAS promises that the network overhead will not be more than other traditional dynamical resource allocations.

## 7 Conclusion

In this paper, we present VRAS, a resource allocation system that dynamically distributes virtualized computing resources among VMs on a local host. VRAS is designed to efficiently allocate resources under light or moderate workloads on host. It leverages Sensor running in the VM to monitor the resource usages and send request signals when the resource requirements change. Two strategies are implemented in VRAS to allocation processor and memory resources. The processors are handled at both fine-grain and coarse-grain level, and the memory resources are scheduled based on the real memory usage. We conduct experiments on different applications, results show that our VRAS performs very well on both computing-intensive and memory consuming applications, 21 and 69 % better than the typical resource allocation on processor and memory resources, respectively. The results on a web application also prove the effectiveness of VRAS. As a lightweight resource allocation system, the network overhead of VRAS in all experiments does not exceed 0.3 kbps.

VRAS have proven good accuracy and agility on resource allocation, especially for light workloads. Next we plan to study the fairness of the request driven mechanism used in VRAS, which is important in heavy workload environment. To address that problem, more information about the inside world of VM would be analyzed and sent out by Sensor. Another interesting challenge is the I/O resource allocation. Our approach may take advantage of its low cost on network.

**Acknowledgments** The research is supported by National Science Foundation of China under Grant No. 61073024 and 61232008. It is also supported by National 863 Hi-Tech Research and Development Pro-

gram under Grant No. 2013AA01A213, Outstanding Youth Foundation of Hubei Province under Grant No. 2011CDA086S, and Guangzhou Science and Technology Program under Grant 2012Y2-00040.

## References


1. Smith, J. E., & Nair, R. (2005). The architecture of virtual machines. *IEEE Computer*, *38*(5), 32–38.
2. Rosenblum, M., & Garfinkel, T. (2005). Virtual machine monitors: Current technology and future trends. *IEEE Computer*, *38*(5), 39–47.
3. Wang, X., Sang, Y., Liu, Y., & Luo, Y. (2011). Considerations on security and trust measurement for virtualized environment. *Journal of Convergence (JoC)*, *2*(2), 19–24.
4. Luo, H., & Shyu, M. L. (2011). Quality of service provision in mobile multimedia—A survey. *Human-centric Computing and Information Sciences (HCIS)*, *1*, 5. doi:10.1186/2192-1962-1-5.
5. Xie, X., Jiang, H., Jin, H., Cao, W., Yuan, P., & Yang, L. T. (2012). Metis: A profiling toolkit based on the virtualization of hardware performance counters. *Human-centric Computing and Information Sciences (HCIS)*, *2*, 8. doi:10.1186/2192-1962-2-8.
6. Nelson, M., Lim, B. H., & Hutchins, G. (2005). Fast transparent migration for virtual achines. *Proceedings of the 2005 USENIX annual technical conference (USENIX 2005)* (pp. 391–394). Anaheim, USA.
7. Clark, C., Fraser, K., Hand, S., Hanseny, J. G., July, E., & Limpach, C., et al. (2005). Live migration of virtual machines. *Proceedings of the 2nd ACM/USENIX symposium on networked systems design and implementation (NSDI 2005), 2* (pp. 273–286). Boston, USA.
8. Abdelzaher, T., & Lu, C. (2000). Modeling and performance control of internet servers. *Proceedings of the 39th IEEE conference on decision and control (ICDC 2000), 3* (pp. 2234–2239). Sydney, Australia.
9. Anderson, M., Kihl, M., & Robertsson, A. (2003). Modeling and design of admission control mechanisms for web servers using non-linear control theory. *Proceedings of the conference on performance and control of next-generation communication networks (ITCom 2003), 5244* (pp. 53–64). Orlando, USA.
10. Lu, C., Abdelzaher, T. F., Stankovic, J., & Son, S. (2001). A feedback control approach for guaranteeing relative delays in web servers. *Proceedings of the 7th IEEE real-time technology and applications symposium*, 51–62, Taipei.
11. Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S., & Tilbury, D. M. (2002). Mimo control of an apache web server: Modeling and controller design. *Proceedings of the 2002 American control conference (ACC 2002), 6* (pp. 4922–4927). Minneapolis, USA.
12. Goel, N., & Shyamasundar, R. K. (2012). An executional framework for BPMN using Orc. *Journal of Convergence (JoC)*, *3*(1), 29–36.
13. Braham, P., Draconic, B., Fraser, K., Hand, S., Harris, T., & Ho, A., et al. (2003). Xen and the art of virtualization. *Proceedings of the 19th ACM symposium on operating systems principles (SOSP 2003, 37(5)* (pp. 164–177). New York, USA.
14. Whitaker, A., Shaw, M., & Gribble, S. D. (2002). Scale and performance in the denali isolation kernel. *Proceedings of the 5th symposium on operating systems design and implementation (OSDI 2002), 36* (pp. 195–209). Boston, USA.
15. Liu, X., Zhu, X., Singhal, S., & Arlitt, M. (2005). Adaptive entitlement control of resource partitions on shared servers. *Proceedings of the 9th IFIP/IEEE international symposium on integrated network management (IM 2005)* (pp. 163–176). Nice, France.
16. Wang, Z., Zhu, X., & Singhal, S. (2005). Utilization and slo-based control for dynamic sizing of resource partitions. *Proceedings of the 16th IFIP/IEEE international workshop on distributed system: Operations and management (DSOM 2005), 3775* (pp. 133–144). Barcelona, Spain.
17. Zhu, X., Wang, Z., & Singhal, S. (2006). Utility driven workload management using nested control design. *Proceedings of the 2002 American control conference (ACC 2006)* (pp. 6033–6038). Minneapolis, USA.
18. Padala, P., Shin, K., Zhu, X., Uysal, M., Wang, Z., Singhal, S., et al. (2007). Adaptive control of virtualized resources in utility computing environments. *Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on computer systems 2007(EuroSys 2007, 41(3)* (pp. 289–302). Lisbon, Portugal.
19. Zhang, Y., Bestavros, A., Guirguis, M., Matta, I., & West, R. (2005). Friendly virtual machines: Leveraging a feedback-control model for application adaptation. *Proceedings of the 1st ACM/USENIX international conference on virtual execution environments (VEE 2005)* (pp. 2–12). Chicago, USA.
20. Qian, H., Miller, E., Zhang, W., Rabinovich, M., & Wills, C. E. (2007). Agility in virtualized utility computing. *Proceedings of the 3rd international workshop on virtualization technology in distributed computing (VTDC 2007)* (pp. 1–8). Reno, USA.

21. Wood, T., Shenoy, P., Venkataramani, A., & Yousif, M. (2007). Black-box and gray-box strategies for virtual machine migration. *Proceedings of the 4th USENIX symposium on networked systems design and implementation (NSDI 2007)* (pp. 229–242). Cambridge, USA.
22. Cherkasova, L., Gupta, D., & Vahdat, A. (2007). Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review*, *35*(2), 42–51.
23. Credit Scheduler. http://wiki.xensource.com/xenwiki/CreditScheduler
24. Waldspurger, C. A. (2002). Memory resource management in VMware ESX Server. *Proceedings of the 5th symposium on operating systems design and implementation (OSDI 2002), 36* (pp. 181–194). Boston, USA.
25. Shi, X., Tan, H., Wu, S., & Jin, H. (2008). VNIX: Managing virtual machines on clusters. *Proceedings of Japan-China joint workshop on frontier of computer science and technology (FCST 2008)* (pp. 155–162). Nagasahi, Japan.
26. Super PI. http://www.superpi.net
27. RUBiS. http://rubis.ow2.org

## Author Biographies

**Hai Jin** was born in 1966. He received his Ph.D. degree in computer engineering from HUST in 1994. Currently, he is a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST. His research interests include cloud and grid computing, P2P computing, and computing system virtualization. He is now Dean of the School of Computer Science and Technology at HUST and a senior member of IEEE and ACM.

**Wei Gao** was born in 1978. He is a Ph.D. candidate in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST). His research interests include computing system virtualization, distributed parallel computing and cloud computing.

**Song Wu** was born in 1975. He received the Ph.D. degree in computer science from Huazhong University of Science and Technology (HUST) in 2003. Currently, he is a professor at SCTS/CGCL of HUST. His research interests include cloud computing, grid computing and system virtualization. He is the director of Parallel and Distributed Computing Institute at HUST.

**Xuanhua Shi** was born in 1978. He received the Ph.D. Degree in computer architecture from Huazhong University of Science and Technology (HUST) in 2005. Currently he is an associate professor at HUST. His research interests include cluster and grid computing, trusted computing, computing system virtualization, and cloud computing. He is a member of ACM and China Computer Federation.