MIDDLEWARE FOR AUTONOMOUS RECONFIGURATION OF VIRTUAL MACHINES

By

Qingling Wang

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

____
Carlos Varela, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2011
(For Graduation August 2011)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my advisor, Prof. Carlos Varela first here. He gives me a lot of helpful and constructive opinions on the experiments and analysis. I also want to thank Shigeru Imai. He also helps me a lot with my research, especially for some Linux kernel stuff. Last, I would like to thank my boyfriend, who gives me a lot of moral support.

ABSTRACT

Cloud computing brings significant benefits for service providers and service users because of its characteristics: *e.g.*, on demand, pay for use, scalable computing. Virtualization management is a critical component to accomplish effective sharing of physical resources and scalability. Existing research focuses on live Virtual Machine (VM) migration as a VM consolidation strategy. However, the impact of other virtual network configuration strategies, such as optimizing total number of VMs for a given workload, the number of virtual CPUs (vCPUs) per VM, and the memory size of each VM has been less studied. This thesis presents specific performance patterns on different workloads for various virtual network configuration strategies. We conclude that, for loosely coupled CPU-intensive workloads, memory size and number of vCPUs per VM do not have significant performance effects. On an 8-CPU machine, with memory size varying from 512MB to 4096MB and vCPUs ranging from 1 to 16 per VM; 1, 2, 4, 8 and 16VM configurations have similar running time. The prerequisite of this conclusion is that all 8 physical processors be occupied by vCPUs. For tightly coupled CPU-intensive workloads, the total number of VMs, vCPUs per VM and memory allocated per VM become critical for performance. We obtained the best performance when the ratio of total number of vCPUs to processors is 2. Doubling memory size on each VM, for example from 1024MB to 2048MB, brings at most 15% improvement of performance when number of VMs is greater than 2. Based on the experimental results, we propose a framework and a threshold-based strategy set to dynamically refine virtualization configurations. The framework mainly contains three parts: *resources monitor*, *virtual network configuration controller* and *scheduler*, which are responsible for monitoring resource usage on both virtual and physical layers, controlling virtual resources distribution, and scheduling concrete reconfiguration steps respectively. Our reconfiguration approach consists of four strategies: VM migration and VM malleability strategies, which are at global level, vCPU tuning and memory ballooning, which are at local level. The strategies evaluate and trigger specific reconfiguration steps (for example, double the number of vCPUs on each VM) by comparing current allocated resources and corresponding utilizations with expected values. The evaluation experimental results of threshold-based strategy show that reconfiguration in global level works better for tightly coupled CPU-intensive workloads than for loosely coupled ones. Local reconfiguration including dynamically changing number of vCPUs and memory size allocated to VMs, improves the performance of initially sub-optimal virtual network configurations, even though it falls short of performing as well as the initially optimal virtual network configurations. This research will help private cloud administrators decide how to configure virtual resources for a given workload to optimize performance. It will also help service providers know where to place VMs and when to consolidate workloads to be able to turn on/off Physical Machines (PMs), thereby saving energy and associated costs. Finally it let service users know what kind of and how many VM instances to allocate in a public cloud for a given workload and budget.

1. Introduction

Cloud computing brings significant benefits for both service providers and service users. For service users, they pay the computing resources only on demand and without worrying about hardware, software maintenance or upgrade. For service providers, with VMs, they can shrink or expand the utilization of physical resources based on workloads' requirements. Therefore, it is possible for providers to make higher profit without affecting users' satisfaction.

Thus, it is worthy to investigate more deeply into adaptive virtual network management techniques. A high ratio of VMs to PMs enables sharing of resources yet guaranteeing isolation between workloads. In contrast to physical resources, VMs provide dynamic control on their system settings, *i.e.*, vCPUs, virtual memory. It is possible to reallocate resources for each VM when necessary. Mature virtualization infrastructures, like Xen and VMware, provide resource modulation on VMs. For example in Xen, you can statically set memory size, number of vCPUs for each VM in the configuration file when starting a VM; you can also change these settings for a specific VM during runtime.

Although it is known that resource reallocation is possible in virtual infrastructure, no specific data shows how these settings impact workloads' performance. In previous preliminary paper [28], we studied the impact of VM granularity (which affects VM/PM ratio) on two categories of workloads. Our performance evaluation indicated that, for tightly coupled computational workloads, VM granularity indeed affects the performance significantly; by contrast, the impact of VM granularity on the performance of loosely coupled network intensive workloads is not critical. This thesis continues the research on the impact of VM granularity on performance but also considers the impact of different virtual network configurations, namely, the number of vCPUs per VM and the allocated virtual and physical memory to each VM. In other words, we tackle the following problem: given a class of workload, what kind of virtual network configuration (*i.e.*, number of VMs, number of vCPUs per VM, memory size of each VM) optimizes the workload's performance and the utilization of physical resources?

Once we get the concrete relationship between the virtual network configurations and workloads'performance, it is good to build a real-time middleware to monitor and reconfigure the virtualization settings of a cloud environment. Such kind of middleware would benefit both public cloud providers and private cloud owners a lot by providing better performance and resource utilizations than a static configuration.

1.1 Cloud Computing

Cloud computing is an emerging distributed computing paradigm that promises to offer cost-effective scalable on demand services to users, without the need for large up-front infrastructure investments [3]. Through the use of virtualization, cloud computing provides a back-end infrastructure that can quickly scale up and down depending on workload [2].

From the view of users, it has the following outstanding advantages:

- Enable users to access data everywhere.
- Provide on-demand services.
- a more reliable and flexible architecture, more chances to guarantee QoS.

Ian Foster et al. give a more detailed definition for cloud computing [8]. They also gives a comprehensive introduction and comparison for cloud computing and grid computing, which helps people to get a better understanding about these two paradigms. Kondo et al. give a quantified cost-benefits comparison between cloud computing and desktop grids application, which also helps make a more clear view of cloud [13].

All the organizations are impressed with the computing, storage and all such powerful abilities of cloud computing systems. However, cloud computing is not a universal method or architecture. It may give a low performance under some inappropriate virtual configurations. The main disadvantages can be summarized as follows:

- It is hard to evaluate exactly how many compute instances, what kind of instance and how many data transfers will be needed.
- The coordination between different virtual machines may degrade the performance.
- In workloads' level, it is nontrivial to separate the data into independent data set.
- For individual customer or some small group, it is still very expensive to use the public cloud computing services.

The first part in this thesis is aiming to overcome the above shortcomings and give some light to both service providers and service users to make better use of cloud computing.

1.2 Virtual Network Configurations

The virtual network configuration includes four parts: (1) VM migration (2) VM malleability (3) vCPUs tuning and (4) memory ballooning.

A lot of work and research has been done on VM migration [9], [15], [24]. By definition, it means migrating a VM from one host machine to a target machine, aiming to make a good use of the physical resources in the whole environment and improve performance for upper applications.

VM malleability referring to VMs split and merge, is more complex than vCPUs tuning and memory ballooning. Given a workload, an initial number of VMs will be assigned to run it. To do VM splitting, we have to move parts of tasks to new VMs; to do VM merging, we have to consolidate the sub tasks. It is easy to do VM split and merge for loosely coupled workloads because the subtasks are relatively independent between each other; and you can move them to anywhere you want. For tightly coupled workloads, we have to consider the communication, the connection between subtasks; and promise the tightly coupled, frequently communicated subtasks would move to the same VM. Therefore, this procedure will more or less influence the workloads' performance. The VM number controller in our framework will check whether there is any VM malleability instruction for every 1% work done and it will execute above refining procedures if yes. The framework will consider to consolidate more relative subtasks together onto the same VM.

For each VM, the number of virtual CPUs (vCPUs) and virtual memory size are configurable. vCPUs tuning means to increase or decrease the number of vCPUs of VMs; memory ballooning means expand or shrink the allocated memory size of VMs. We have to specify the maximum number of vCPUs and maximum size of virtual memory the VM could have in the configure file. Xen provides special commands, '`vcpu-set`' and '`mem-set`', to change vCPUs and memory size at all times including application run time; the only restriction is that the new vCPUs number and memory size can not exceed the corresponding maximum values in the configure file.

This thesis is organized as follows. Chapter 2 describes the impact of virtualization strategies on performance. The autonomous framework to implement VM malleability is proposed in Chapter 3. Chapter 4 describes the virtual machine reconfiguration strategies to dynamically and autonomously adjust the virtual network configurations. In Chapter 5, we evaluate the performance evaluation of the strategies described in Chapter 4. Related research is discussed in Chapter 6. Finally, Chapter 7 concludes the thesis and discusses future work directions.

2. Impact of Virtualization on Cloud Computing Performance

This chapter aims to find the relationship between virtual network configuration strategies and performance of categories of workloads. We have built a testbed for experimentations, performance evaluation, and strategies defining for virtual network configurations.

2.1 Testbed Setup

In order to minimize the influence of other factors, for example bandwidth, and to concentrate on the relationship between the virtual and physical layer, we deploy most of out experiments on one server machine. The server is equipped with a quad-processor, dual-core Opteron. The Opteron processors run at 2.2GHz, with 64KB L1 cache and 1MB L2 cache. The machine has 32GB RAM. The hypervisor is Xen-3.4.1 and the operating systems of the VM instances are Linux 2.6.18.8 and Linux 2.6.24-29.

2.2 Workloads

We study three different categories of workloads: the first one is *Stars Distance Computing*, a loosely coupled CPU-intensive workload; the second is *Heat Distribution* problem, a tightly coupled CPU-intensive workload. They are both developed in SALSA [26], using actors[1] to perform distributed sub tasks. Load-balancing is taken into account in our experiments such that each actor is responsible for the same amount of work. We also give a brief analysis for a loosely coupled external network-intensive workload - a Web 2.0 system. Table 2.1 shows the corresponding categories of the three workloads.

2.2.1 Stars Distance Computing

The goal of this problem is to analyze three-dimensional data from the Sloan Digital Sky Survey, in particular, stars from the MilkyWay galaxy. The program

Table 2.1: Categories of workloads

	Stars	Heat	Web 2.0 System
Loosely Coupled	$\surd$		$\surd$
Tightly Coupled		$\surd$	
CPU-Intensive	$\surd$	$\surd$	
Network-Intensive			$\surd$

computes the closest and farthest neighbor stars, which are the set of pairs of stars that minimize and maximize pairwise distance respectively; the ideal hub stars, which minimize the maximal distance to any other star; the ideal jail stars, which maximize the minimal distance to any other star and the ideal capital stars, which minimize the average distance to all other stars.

In this problem, each actor is responsible for computing a certain number of stars and it holds the immutable data it needs, and therefore the actors are independent from each other.

2.2.2 Heat Distribution

This problem simulates heat distribution on a two-dimensional surface. Suppose the surface is a square grid and it has some size (for example, 122 by 122). The initial temperature is 100 degrees for some coordinates (for example, $\{(y, 0)|30 < y < 91\}$) and 20 degrees for all other points in the surface. The temperature for a grid element at a given time step is the average of the grid element's neighbors at the previous time step (boundary elements remain constant over time.) This workload simulates heat transfer for certain time steps to approximate equilibrium temperatures.

In this problem, each Actor is in charge of specific sub blocks of the grid and all the Actors have to communicate with others to get the neighbor temperatures to update their boundary elements.

2.2.3 Web 2.0 System

The web system is built as a distributed system and serves to process continuous requests from clients. Since the server nodes in the system are independent from each other, the web system is a loosely coupled external network intensive workload.

2.3 Benchmark Architecture

Figure 2.1 shows the architecture of our performance evaluation system. We use Xen as our *virtual machine manager*. The *virtual network configuration controller* is responsible for dynamically controlling and refining the total number of VMs , vCPU number per VM and memory size allocated to each VM . The *resource usage monitor*, running on both virtual and physical layers, periodically collects CPU, memory consumption information.

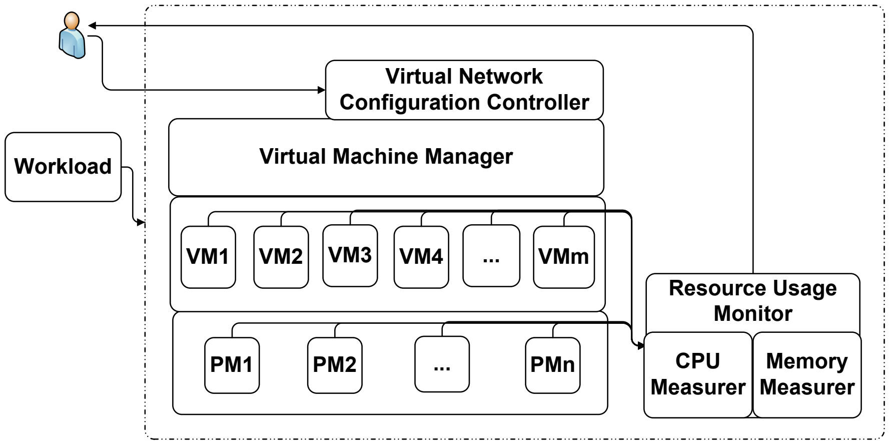


Figure 2.1: Benchmark architecture

2.4 Experimental Design

The overall objective of our experiments is to quantify the slowdown or improvement of a given workload when the virtual network configurations are changed in a fixed physical environment. Private cloud owners, service providers and service users will all benefit from our experiments. *Private cloud owners* have the rights to change virtual network configurations at any time. They will benefit by knowing how to configure and place VMs for limited physical resources, when to consolidate VMs to be able to turn off servers to save energy without effecting performance.

Service providers will benefit by knowing the proper time and way to consolidate VMs, thus to maximize profit but still keeping well-provisioned VMs to guarantee service quality. finally *Service users* will know what kinds of (for example, small, or large? high-memory or high-CPU?) and how many VM instances maximize a workload's performance under a given budget.

In order to find out the answers, we conduct overlapping experiments on the workloads with different virtual network configurations:

- VMs and vCPUs

  In this experiment, we make the memory size on each VM as a constant, change total number of VMs from 1 to 16 and change vCPU number of each VM from 1 to 16. We collect the workloads' performance on every pair of VMs and vCPUs per VM.

- VMs and virtual memory size

  Similar to the first experiment, we make the vCPU on each VM as a constant, for example 4, but change total number of VMs from 1 to 16 and change virtual memory from 512MB to 4096MB. We collect the workloads' performance on every pair of VMs and Virtual Memory per VM.

- vCPUs and virtual memory size

  Last, we make total number of VMs in our experimental environment as a constant, for example 4, change vCPU and memory size on each VM as previous. We collect the workloads' performance on every pair of vCPU and Virtual Memory setting.

2.5 Results and Analysis

Our experimental results show that for loosely coupled CPU-intensive workloads, the number of vCPUs and memory size have no significant impact on performance. The prerequisite is the total number of vCPUs has to be greater than or equal to the physical processors. For tightly coupled CPU-intensive workload, the virtualization settings become critical for performance. For loosely coupled network intensive workloads, VM consolidation could be more efficiently applied.

2.5.1 Loosely Coupled CPU-Intensive Workload

Figure 2.2 shows the performance change on different number of VMs with different number of vCPUs per VM. When the total number of vCPUs is less than the number of physical processors, either increasing number of VMs or number of vCPUs per VM will reduce the running time significantly; increasing number of vCPUs of VMs alone cannot bring down running time as quickly as increasing number of VMs.

After physical processors are all occupied, increasing number of VMs or vCPUs per VM has no significant influence on performance. Instead, too many vCPUs (the total number of vCPUs is greater than the number of actors) will decrease the performance.

Performance on different number of VMs and different memory size of each VM shows in Figure 2.3. In this experiment, we fix the number of vCPUs on each VM as 4. Figure 2.3 indicates that adding VM instances or allocating more virtual memory to each VM will not necessarily improve performance. On the contrary, it may raise VM configuration exceptions and cause a downtime if servers do not have enough physical memory to allocate. In our experiment, memory becomes a bottleneck when number of VMs gets to 8.

Next, we will investigate how different number of vCPUs and different size of memory per VM affect the performance of loosely coupled CPU-intensive workloads. We fix the number of VMs in our environment as 4, Figure 2.4 presents the results.

The results indicate that allocating larger memory to VMs will not necessarily improve performance; or we can say that smaller memory size of each VM will not worsen performance. Adding more vCPUs per VM after all physical CPUs are occupied is not workable either. It will worsen performance on the contrary, though just a little bit.

For loosely coupled CPU-intensive workloads, we conclude that enough number of VMs or vCPUs has to be guaranteed to fully utilize physical processors.

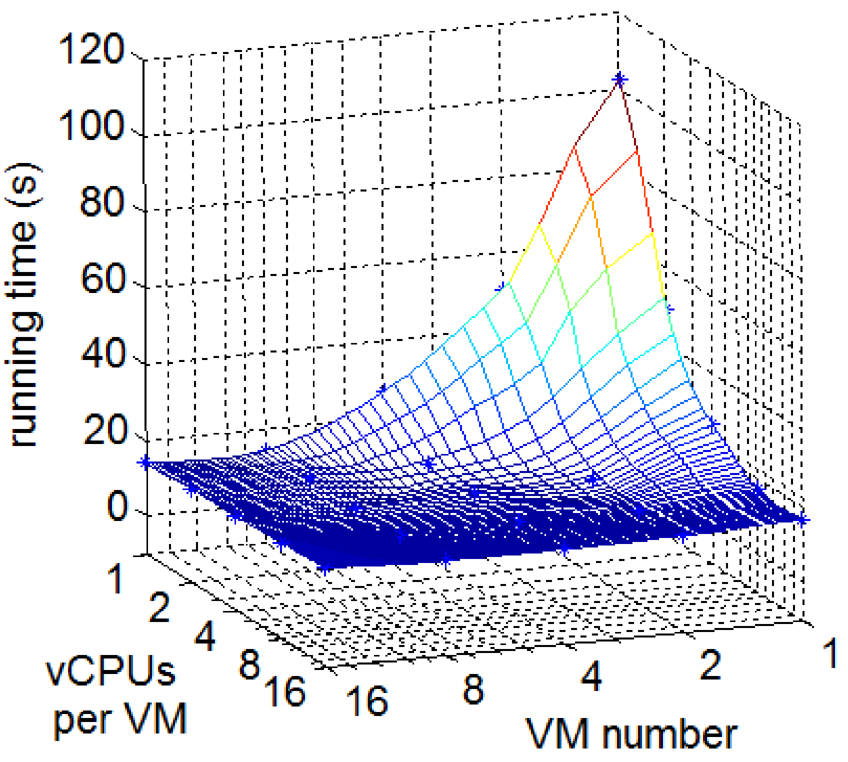


Figure 2.2: Performance of *Stars* running on different vCPUs, VMs settings (512MB memory per VM

However, small number of VMs with large number of vCPUs per VM is not preferred since it cannot produce the best performance. The memory size allocated to VMs, as long as it is enough to run the workloads, basically has no impact on performance. CPU utilization is the main determining factor on performance.

These valuable results provide an important reference for both service providers and service users in cloud environments. For service users, if the workload is loosely coupled CPU-intensive, it is waste of money to request high-memory or high-CPU instances. It is more worthy to request more VMs. Service providers will benefit by knowing where, when and how to place VMs: first, the administrators need to guarantee all physical processors will be occupied when place or consolidate VMs; second, consolidation is only feasible when memory is enough. Private cloud owners can learn how to configure virtual resources from the results for the loosely coupled

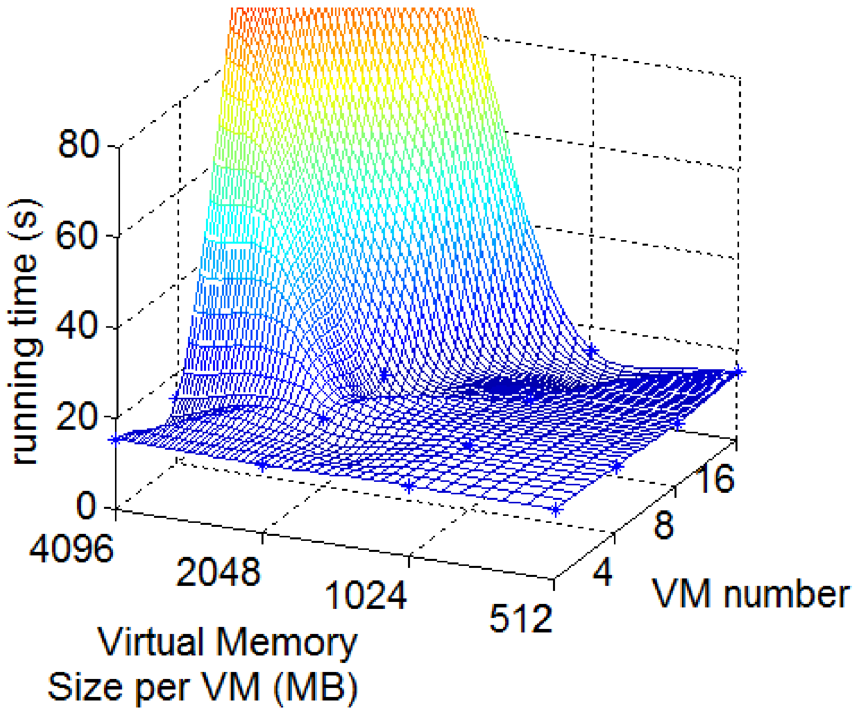


Figure 2.3: Performance of *Stars* running on different VMs, virtual memory settings (4vCPUs per VM)

computational workloads. First, ensure that the ratio of vCPU to CPU is greater than 1 to guarantee a good performance; second, shrink memory to save energy and resource consumption is feasible.

2.5.2 Tightly Coupled CPU-Intensive Workload

We also conduct three different but overlapping experiments on Heat Distribution workload. The virtual memory on each VM is fixed to 1024MB. Figure 2.5 shows the performance change on different number of VMs and vCPUs. Different from loosely coupled CPU-intensive workload, there exists a specific configuration to obtain the best performance.

We assign equal amount of actors to each VM for this workload. The actors first compute new temperatures for their own responsible scope, then communicate with other surrounding actors to update the boundary. The number of VMs becomes critical. Less VMs means more intense CPU competitions between actors on the

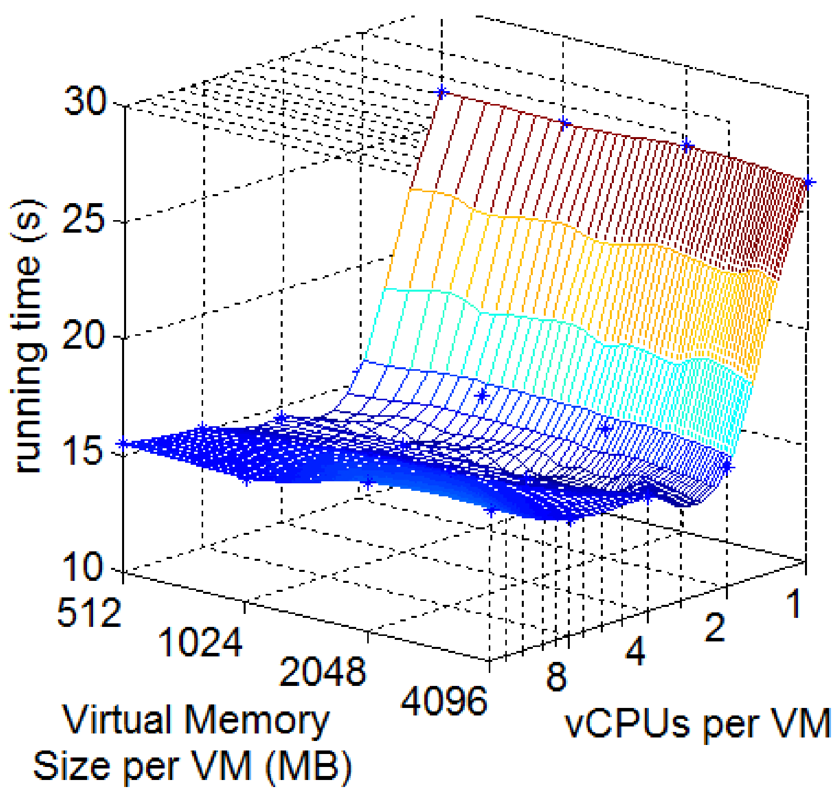


Figure 2.4: Performance of *Stars* running on different vCPUs , virtual memory settings (on 4VMs)

same VM; more actors per VM means more memory needed to save intermediate-data. As a result, the computation step of each actor will become slow, such slow computation plus limited memory will further influence the next communication step; then slow communications between actors will affect the next computation step in return. In one word, a high ratio of actors to VMs is dangerous because it will lead to a very low performance. For example, in our experiment when we have 2 VMs, no matter how many vCPUs per VM we have, running time is always very long, even hang-up. Too many VMs or to say a low ratio of actors to VMs is also inadvisable. In such case, context switching will become bottleneck because of more communication between different VMs needed. For vCPU setting, it is also not trivial. Less vCPUs per VM can not make full use of physical CPUs but more vCPUs will make actors unable to fully utilize vCPUs. In our experiment, when setting the total number of vCPUs 2 times physical processors, it presents the best running time.

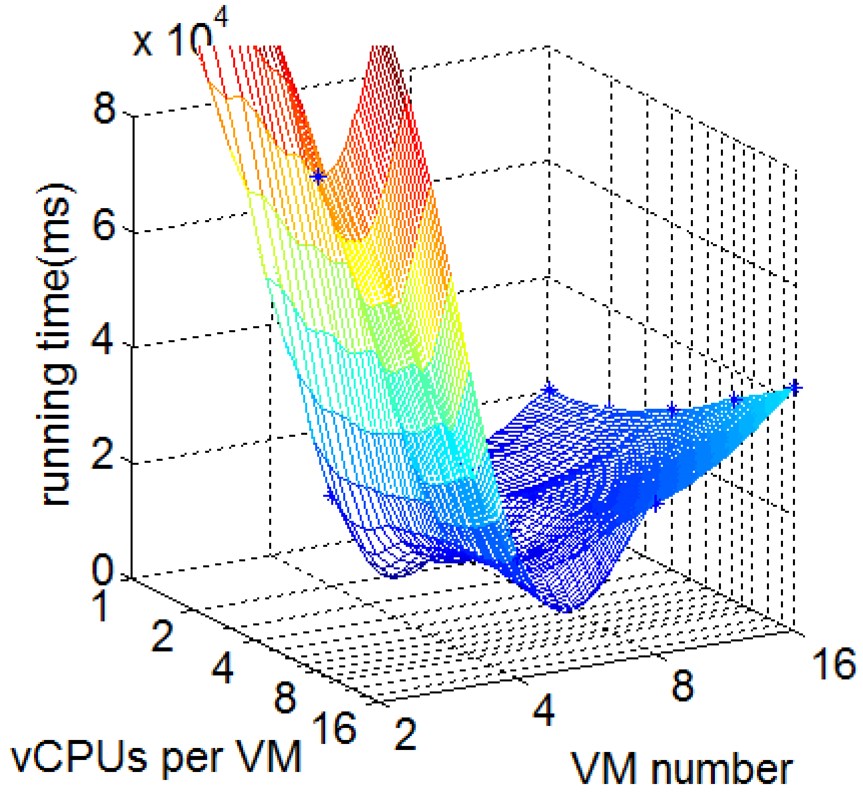


Figure 2.5: Performance of *Heat* running on different VMs with varied number of vCPUs (1024MB memory per VM)

The goal of this experiment is to investigate the impact of different number of VMs and different memory size per VM on workloads' performance. To get rid of the influence of CPU, we set the number of vCPUs on each VM as 4. Thus, no idle physical processors exist in this serial of experiments.

Figure 2.6 indicates the relationship between number of VMs and memory size per VM is subtle. Higher memory size per VM is generally better except when number of VMs in the environment grows. The performance improvement caused by increasing memory size is inconsequential, especially when the number of VMs reaches to a certain extent. For example, in our experiment when VM number is greater than 2, performance improvement by only increasing virtual memory on every VM has been around 10%.

Memory becomes a bottleneck when the number of VMs is small. Less VMs means more actors per VM thus more memories are needed for a VM to save all data and intermediate-results. The graph shows that 2 VMs with 512MB or 1024MB memory per VM results in very bad performance, several times worse than other configurations; allocating more memory to each VM to 2048MB or larger, the running time will decrease to some ideal point. However, keeping a large memory for every VM is not reasonable either. The reasons are as follows. First, as number of VMs grows, memory size becomes uncritical and performance improvement is not significant by increasing memory size on VMs; second, creating new VMs becomes unworkable due to lack of free physical memory; last, a VM with large memory is not recommended for migration. In our experiment, when we have 8 VMs, the largest memory we can allocate to each VM is 2048MB; while when having 16 VMs, the largest memory can be allocated is 1024MB.

In a word, it is safe to allocate a large memory on each VM; but when number of VM grows, memory ballooning would be needed in order to keep reasonable performance.

The last experiment is about the impact on workloads' performance when given different number of vCPUs and different virtual memory size on every VM. We set the total number of VMs in the system as 4 and the results show in Figure 2.7. As mentioned before, running time becomes really long if the number of vCPUs per VM is small, especially when memory size of VMs is also small. This is because all actors will fall into a vicious circle due to low CPU utilization but high memory consumption. In such case, either larger memory size or more vCPUs per VM will make a improvement.

Figure 2.7 also presents we can find a configuration strategy to give the best performance. In this experiment, 4 vCPUs per VM give the best performance if given the same size of memory on VMs. More vCPUs will bring down performance because of the increased context switches.

Our experimental results provide a transparent way for service users to know what kind of instances they need for tightly coupled CPU-intensive workloads. Given

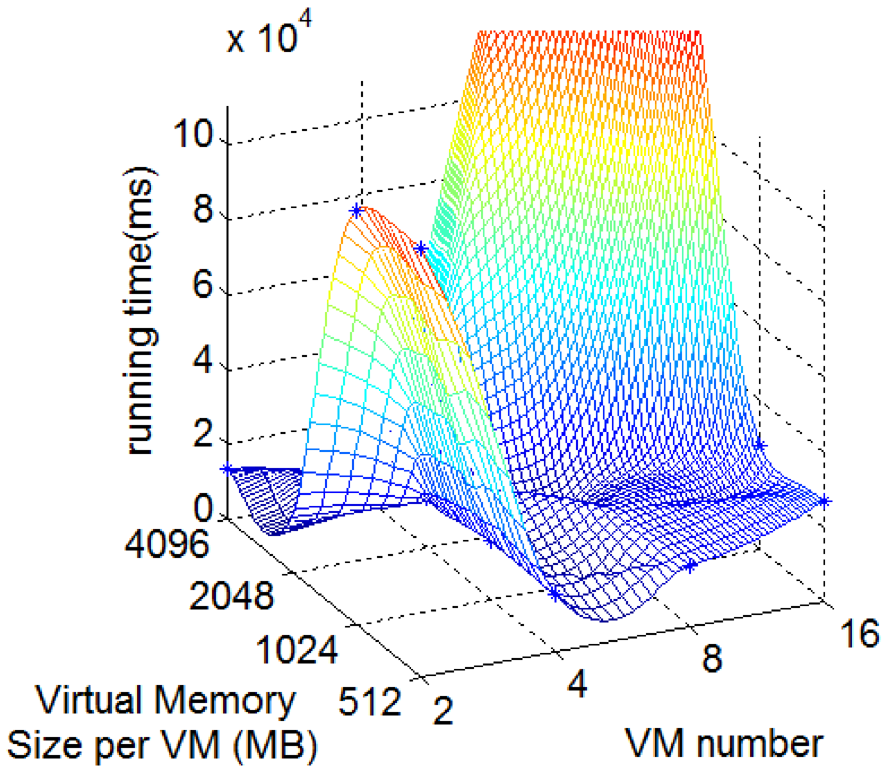


Figure 2.6: Performance of *Heat* running on different VMs with varied virtual memory size (4 vCPUs per VM)

cost budget, large VM instances will not necessarily give better performance; it may not as effective as to request more small instances. Service users should avoid the peek area as shown in our graph. Public cloud providers will benefit by knowing where and how to place and consolidate VMs from the results. Same as before, the administrators need to guarantee all physical processors will be used. Besides, they also need to try not consolidating the VMs with large memory together. As for private cloud owners, they can learn how to configure virtual resources for tightly coupled CPU-intensive workloads. Since the relationship between number of VMs, number of vCPUs per VM and the memory size of VMs is subtle, the private cloud owners have to balance between the configuration strategies. Small number of VMs with small number of vCPUs and small size of memory of each VM should always be forbidden. The number of VMs should be the first thinking in order to stay clear of the peak. Higher memory per VM is generally better but the improvement is not

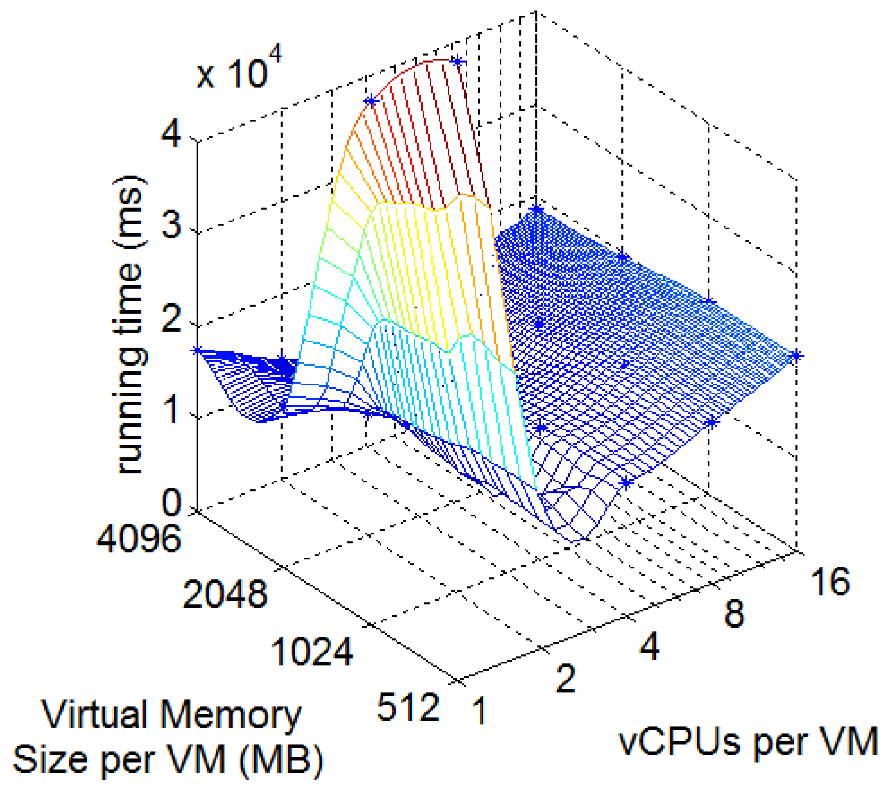


Figure 2.7: Performance of *Heat* running on different vCPUs and virtual memory settings (on 4VMs)

significant. If large number of VMs is needed, the administrator should consider to shrink the memory size on VMs.

2.5.3 Loosely Coupled Network-Intensive Workload

Figure 2.8 shows that VM granularity is not critical for performance of loosely coupled network intensive workloads. It is easy to infer that the VM configurations will not have significant influence on performance either. Therefore consolidation would be more effectively applied for this category of workloads.

2.6 Resource Usage

In order to develop autonomous controllers for refining number of VMs, number of vCPUs and virtual memory size, we need to understand and solve following key problems:

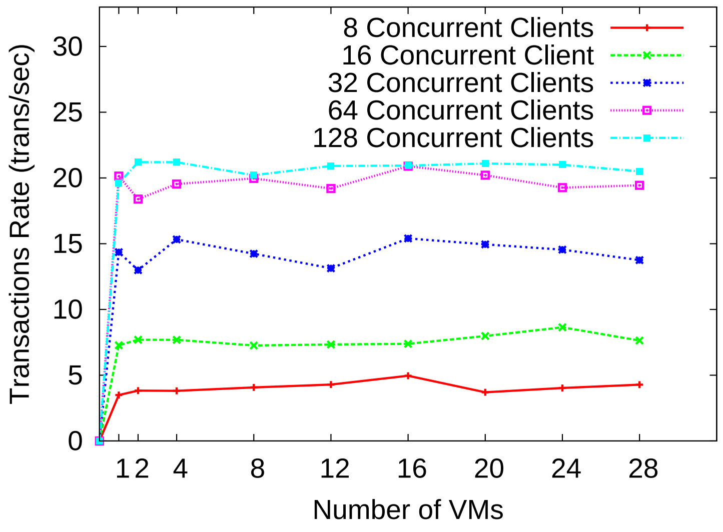


Figure 2.8: The impact of VM granularity on network-intensive workload's performance (results from [28])

- When do we need to reconfigure virtual resources?

  Performing reconfiguration dynamically is not trivial, especially for VM malleability. The actors have to be redistributed to new VMs. Live migration, which will cause a down time [27] during execution, may be needed when number of VMs grows or shrinks. Therefore, it is unreasonable to change the virtual configuration constantly because it will degrade performance instead of improving. To avoid changing settings back and forth, we need to find the proper time to perform the reconfiguration.

- How to module or control virtual resources?

  Even if we have already known the best time to make a change, what specific and effective steps do we need to do? In some cases, allocating more vCPUs is better to have more VMs since it can obtain a reasonable performance but avoiding migration cost. While some times, creating or consolidating VMs is a must to keep applications running. Virtual memory is another important factor needed to consider when performing malleability. There is a subtle relationship between different settings and performance. We need to find or create a module to proceed malleability.

- What is the impact of configuration changes on performance?

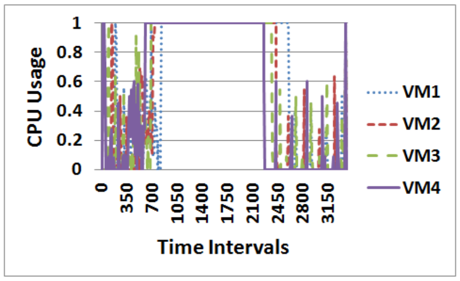
  We have mentioned above, no matter vCPU or virtual memory adjusting or VM number changing, they are not trivial. The aim of virtual network reconfiguration is to obtain benefits for both service provider and service user, but unavoidably, such modulation on virtual resources also bring negative influence on performance. We need to understand the quantitative relationship between without and with performing malleability.

To solve above problems, we need to analyze resources usage on both virtual and physical layer. We developed a sub-module, Resource Usage Monitor which runs on both PMs and VMs, to collect CPU and memory statistics during workloads' execution. These statistics were collected and measured from the `/proc` file system, during that 10ms interval. All experiments were repeated five times for average usage. In our experiments, we collected the usage statistics on all kinds of configuration strategies. However, we cannot present all of them here due to limited space. We choose some typical settings which can sufficiently express the inside key points.

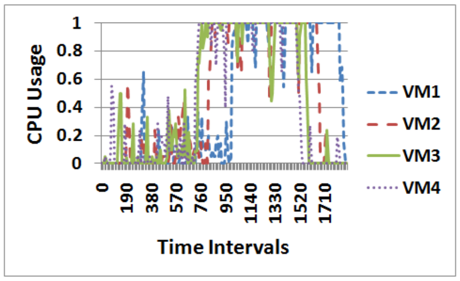
2.6.1 Resource Usage of the Stars Workload

Figure 2.9 displays the measured CPU usage on 4 VMs when the VMs are given different number of vCPUs.

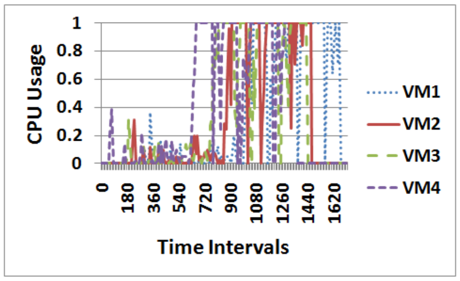
In Figure 2.9(a), CPU utilization stays at 100% for a long time and is relatively high during other time intervals. This demonstrates the VMs have a heavy workload for low number of vCPUs configurations (*i.e.* 1vCPU per VM in our experiments). In Figure 2.9(b), Figure 2.9(c) and Figure 2.9(d), the vCPUs are not as busy as Figure 2.9(a). But the throughput is relatively low for high number of vCPUs per VM configurations; take Figure 2.9(d) for example, the CPU utilization is less than 20% during a long time interval and only get 100% at some time points; the throughput is lower when having 16 vCPUs per VM. Though we have low throughput for high number of vCPUs configurations, the work load for each vCPU becomes relatively



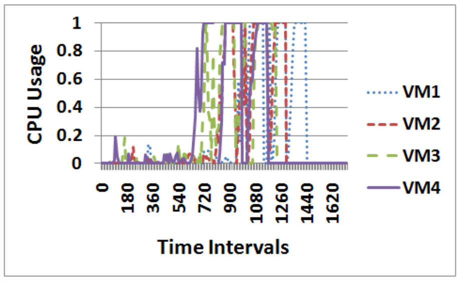
(a) 1vCPU



(b) 2vCPU



(c) 4vCPU



(d) 8vCPU

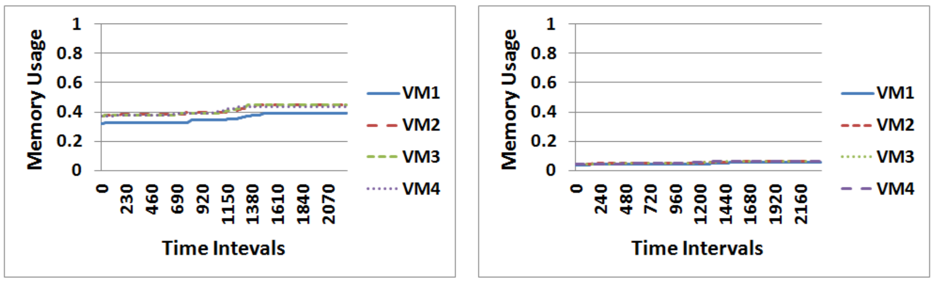
Figure 2.9: CPU usage of *Stars* on different vCPUs settings

light. That is why we get similar performance for different number of vCPUs virtual configurations.

Service providers and private cloud owners should avoid the heavy CPU load situation and make sure the physical processors are all occupied by the VM instances. For service users, they will learn that the VM instances with multi-processors will not necessarily bring better performance given loosely couple computational workloads. The key here is the throughput of the CPUs.

In Figure 2.10, we demonstrate the memory usage (by percent) on 4 VMs given different memory size to each VM.

Figure 2.10(a) has a relatively high memory utilization, greater than 40% for most time. The utilization in Figure 2.10(b), which has a much larger memory on every VM, falls into a very low utilization ($< 10\%$). For the loosely coupled computational workloads, no more additional memory is needed during runtime once all data are loaded into memory at the start. Increasing memory size for each VM thus will not bring any improvement on performance. It is a waste to allocate large memory size for VMs.



(a) 512MB Virtual Memory Size per VM (b) 4096MB Virtual Memory Size per VM

Figure 2.10: Memory usage of *Stars* on different virtual memory settings

Figure 2.10 helps private cloud owners to know when memory ballooning is needed: generally, high memory utilization means ballooning up is needed and low utilization means memory ballooning down is needed. Service providers will benefit by knowing how to consolidate VMs based on memory utilization: for example, consolidating the workloads of low memory consumption and high memory consumption together will improve resource utilization but without influencing performance. The results also let service users know that simply going for high-memory instances are waste of money for loosely coupled computational workloads.

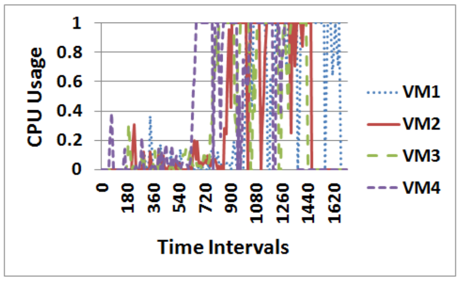
We have discussed the resource usage on different vCPU and virtual memory VM configurations. Given fixed number of vCPUs and memory size per VM, what kind of resource utilization we will have for different number of VMs? Figure 2.11 shows the corresponding CPU, memory utilization on 4 VMs and 8 VMs respectively.

They have similar CPU and memory utilization. Here it tells us that double the resources (including CPU, memory) for loosely coupled computational workloads will not necessarily improve performance, not even to speak double the performance.

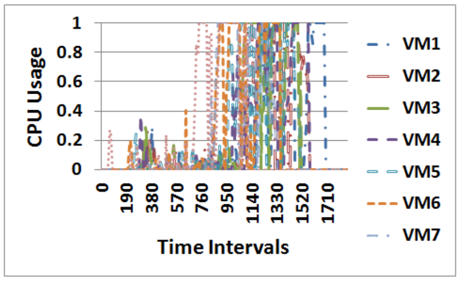
2.6.2 Resource Usage of the Heat Workload

In our experiments, we collect CPU and memory usage statistics on all configurations. we cannot present all of them here due to limited space. We choose some typical settings which can sufficiently express the inside key points.

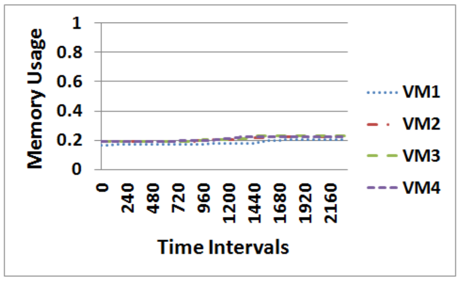
Figure 2.12 displays the measured CPU usage on 4 VMs and each VM has different number of vCPUs.



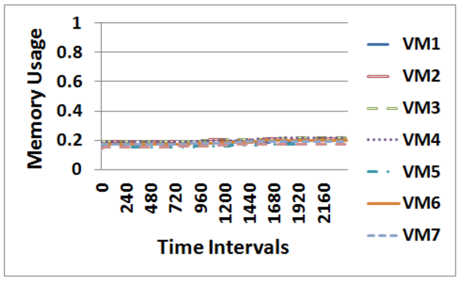
(a) CPU utilization on 4VMs



(b) CPU utilization on 8VMs

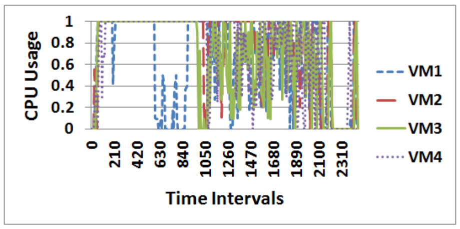


(c) Memory utilization on 4VMs

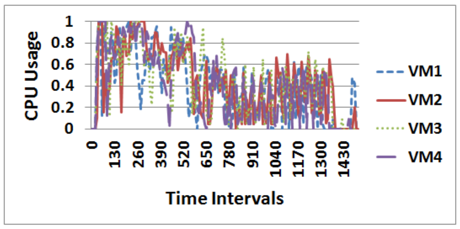


(d) Memory utilization on 8VMs

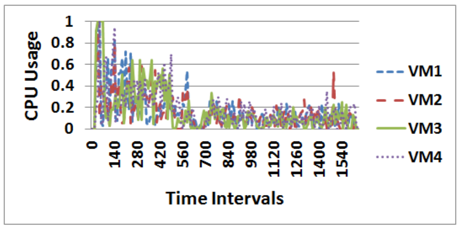
Figure 2.11: CPU and memory usage of *Stars* on different VMs settings



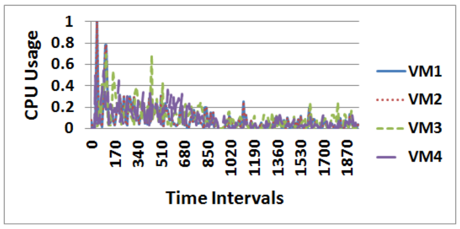
(a) 1vCPU



(b) 4vCPU



(c) 8vCPU



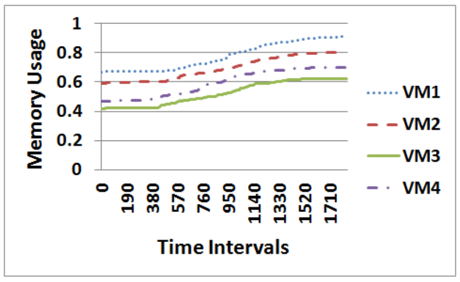
(d) 16vCPU

Figure 2.12: CPU usage of *Heat* on different vCPUs settings

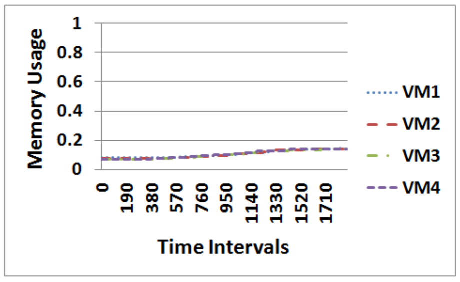
In Figure 2.12(a), we can see that CPU utilization is high (> 90%) for almost all the time. The utilization is generally less than 30% in Figure 2.12(d). This demonstrates the VMs have a very high load for low number of vCPUs settings (*i.e.* 1vCPU per VM in our experiments) , but have a low throughput for high number of vCPUs setting (16vCPUs per VM in our experiments). Figure 2.12(c), which has 8vCPUs per VM has better CPU utilization than Figure 2.12(d), but is still not efficient enough. Figure 2.12(b), the 2vCPUs per VM setting is the best one among others. It has a reasonable utilization (> 60%), not high as Figure 2.12(a) and also not low as 8vCPU or higher setting. 2vCPUs per VM setting is not showing here. It has very similar utilization with Figure 2.12(b).

These resource consumption data can help service providers and private cloud owners know when they need to change their virtualization strategies. If CPU utilization is very high during some time interval, private cloud owners should consider to add more vCPUs for each VM; service providers may need to guarantee no more CPU competition will come. On the other hand, if CPU utilization is very low in a time interval, for private cloud owner, it is better to decrease the number of vCPUs; for service providers, consolidation is feasible in order to save energy.

In Figure 2.13, we demonstrate the memory usage (by percent) on 4 VMs given two different memory size of each VM.



(a) 512MB



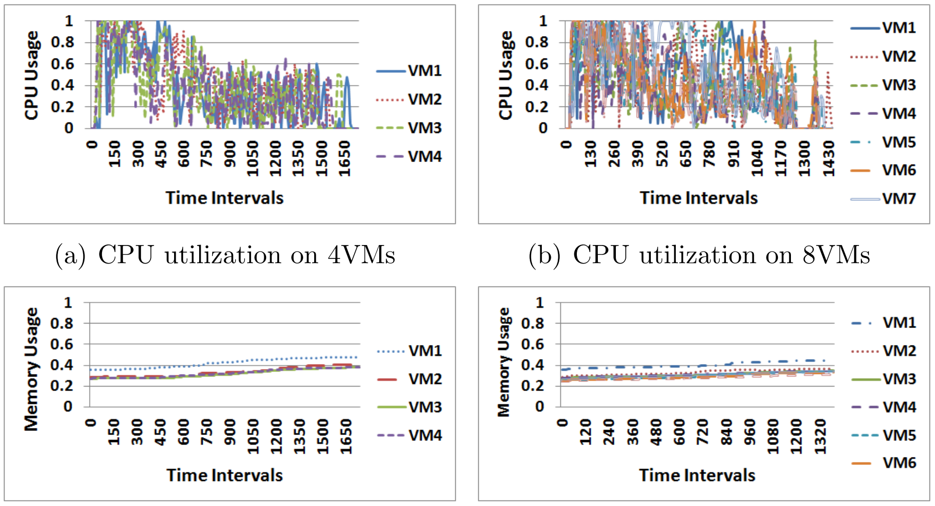
(b) 4096MB

Figure 2.13: Memory usage of *Heat* on different virtual memory settings

Figure 2.13(a) has a relatively high memory utilization, greater than 60% for most time; The utilization in Figure 2.13(b), which has a much larger memory on every VM, falls into a very low level (< 20%). As mentioned before, larger memory is generally better. However, to some extent it is a waste if the memory is too large.

Figure 2.13 helps private cloud owners to know the proper time for memory ballooning. The first concern is the starting point in the graph: if memory utilization is already high at the beginning, it is more likely we need to allocate a larger memory. The second concern is the memory utilization during runtime: if it stays at a high percentage, we need to allocate more memory to VMs. The results also let service users know that large instances are waste of money in some cases.

We have discussed the resource usage on different vCPU and virtual memory configurations. Given fixed number of vCPUs and memory size per VM, what kind of resource utilization we will have for different number of VMs? Figure 2.14 shows the corresponding CPU, memory utilization on 4 VMs and 8 VMs respectively.



(a) CPU utilization on 4VMs (b) CPU utilization on 8VMs

(c) Memory utilization on 4VMs (d) Memory utilization on 8VMs

Figure 2.14: CPU and memory usage of *Heat* on different VMs settings

First, memory is not a bottleneck in these two cases and they have similar utilization. The 8VMs configuration has a better CPU utilization. It is also a proven fact that it improves the performance about 15% by increasing VM number from 4 to 8. Here it tells us we need to weight and balance between resource and performance. We double the resources (including CPU, memory) but get 15% rewards on efficiency.

3. Cloud Operating System Autonomous Framework

Virtualization yields many benefits into modern systems. Virtual machine malleability, which can be cost or performance aware, brings more flexibility, efficiency and transparency into the virtual environment. In this section we will present a common framework to autonomously perform virtual network reconfiguration based on profiled information.

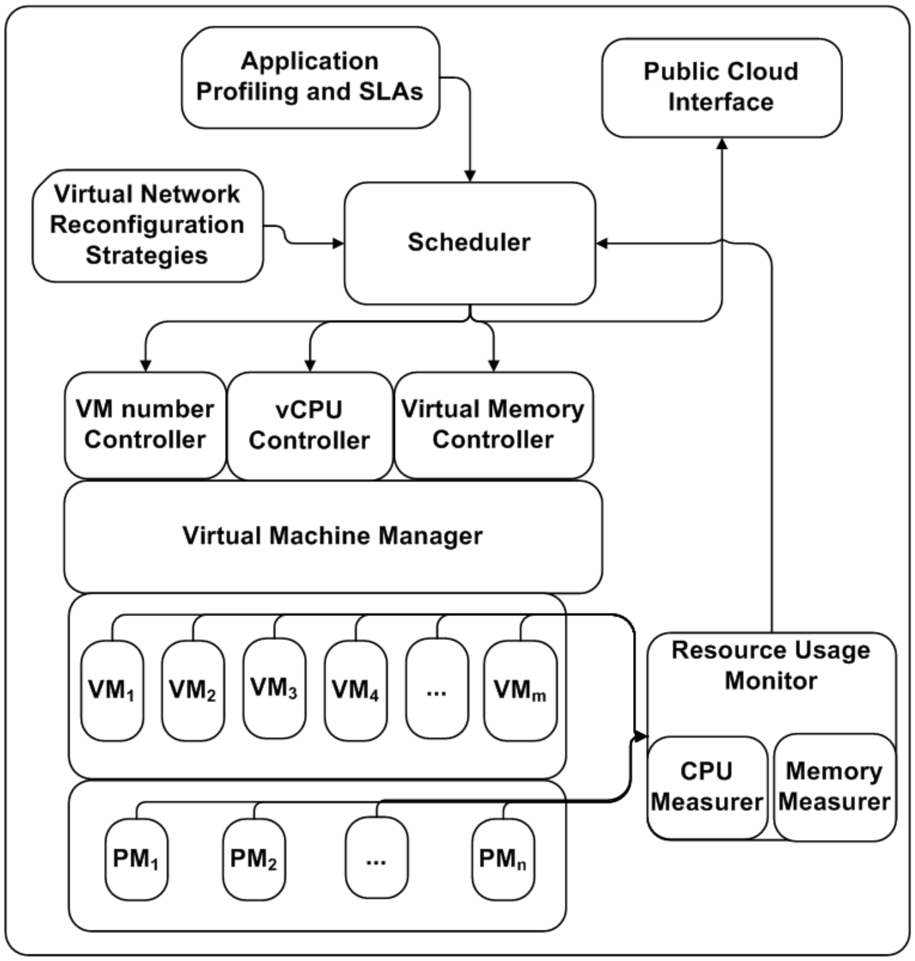


Figure 3.1: Framework of autonomous virtual network reconfiguration

El Maghraoui et al. presented a framework for dynamic reconfiguration of scientific application [16], [18] and [17]. In this thesis, we will focus on the dynamic virtual network reconfiguration on lower level - Virtual Machines.

To accomplish VM malleability, we limit workloads to be reconfigurable (migratable) distributed applications. We have mentioned in the introduction chapter, VM malleability includes VM split and VM merge two directions. To split a VM, we create new VMs and migrate part of the workload to the newly created VMs. To merge VMs, we migrate the workload into a single VM and destroy the empty VMs.

Figure 3.1 shows the common autonomous framework.

3.1 Scheduler

Scheduler is the most important part in the framework. It acts as a master for all the other components. It gathers resource usage information from bottom monitor module and accepts the VM malleability suggestions from the controller layer; Moreover, it takes virtual network reconfiguration strategies and the SLAs as input to make overall decision. After that, it calls the Controllers to perform their corresponding work.

3.2 Virtual Machine Manager

We installed Xen virtualized server to provide virtualization in our environment. Nowadays, Xen itself provides dynamic reconfiguration on VM settings, like vCPU number and memory size. Besides, Xen support live migration to dynamically move one VM from a host machine to a target machine.

Our aim is to implement autonomous VM malleability, however Xen only provides command-line interface to change the settings. This is not what we wanted. We developed a module, called *virtual machine manager* to autonomously perform virtual network reconfigurations. It is a simple wrapper which wraps Xen commands into it meanwhile provides interface to upper level.

We use the commands `vcpu-set` to change vCPU number on VMs; use `mem-set` to change the memory allocations. Each VM is configured with a maximum vCPU number and memory size at the beginning. The virtual machine manager receives instructions from upper controller to call the corresponding commands to change resource configuration dynamically.

3.3 Resource Usage Monitor

The resource usage monitor reads CPU and memory statistics from the `/proc` file system on every VM and PM. Then it will calculate the corresponding utilizations and return them to the scheduler module.

This resource usage monitor module plays as a helper role for main components. It helps to identify whether to perform virtual network reconfiguration based on current resource status.

3.4 Virtual Network Configuration Controllers

We developed three resource controllers. They are independent from each other and are responsible for vCPU, virtual memory size and VM number adjustment respectively.

They gather the information of resource usage from resource usage monitor and determine new VM allocations (or configuration) for next reconfiguration period. The criteria for virtual network reconfiguration is established by the scheduler module. The controller needs to get an affirmative reply from the scheduler to make any change.

3.5 Interface to Public Cloud

This module is optional in the framework. It provides an interface to create VMs in a public cloud, such as Amazon EC2. It is highly possible that our private cloud is not enough to meet application execution requirements. In such case, our scheduler will estimate how many public resources we need and create corresponding VM images to request more VMs. The critical point here is the interrelation between cost and increased performance from this.

4. Virtual Network Reconfiguration Strategies

Our experiments and collected data have proved that proper resource utilization is key to performance. Therefore, it is possible to define a strategy set to trigger virtual network reconfiguration autonomously by monitoring the resources utilization on both virtual and physical layers. In this chapter, we will discuss a simple but effective threshold-based strategy. The thresholds in the strategy set mean the threshold of resource utilization. Table 4.1 shows the summary of our strategy set.

Actually, there is a preferred order for the four strategies, from global strategies to local ones. The VM migration strategy usually comes to the first thought because we want to achieve load balance in a cloud environment, so as to make better use of the physical resources. For a given workload, the number of VMs is the most important factor for performance. Therefore, having the right number of VMs through VM malleability become our second thought. Then we go into the local strategies. The order of vCPUs tuning and memory ballooning is depended on the category of the workload. If it is CPU-intensive, vCPU tuning should be considered first; if it is a memory-intensive workload, the memory ballooning should come first.

For the resources utilization of both virtual and physical layers, we define an expectation value range for it, for example $70\% - 80\%$. If the current resource utilization goes into this range, the framework do nothing and the whole system goes into a temporary stable status; if the resource utilization goes out of the range, namely less than the minimum value of the range or greater than the maximum value of the range, then the framework will be triggered to perform virtual network reconfigurations.

4.1 VM Migration and Fine Tuning Strategy

The system strategy is based on the physical resource consumption data. It helps to make the most correct virtual network reconfiguration direction at the beginning.

Let M_t be the total memory size the servers have and M_o be the memory occupied by operating system. Then $M_r = M_t - M_o$ is the adjustable memory. Let

Table 4.1: Overview of threshold-based strategy

Strategy	Formula	
VM Migration	Memory oriented	CPU oriented
	$VM_n \rightarrow$ (migrate VMs), if $r_M > r_{SME}$	$VM_n \rightarrow$, if $r_C > r_{SCE}$
VM Malleability	Virtual Memory oriented	vCPU oriented
	$VM_n/2$, if $\mu_M < \mu_{E_{min}}$ & $M_n \cong M_l$; $2VM_n$, if $\mu_M > \mu_{E_{max}}$ & $M_n \cong M_h$	$VM_n/2$, if $\mu_V < \mu_{VE_{min}}$ & $V_n \cong V_l$; $2VM_n$, if $\mu_V > \mu_{VE_{max}}$ & $V_n \cong V_h$
Memory Ballooning	Stable State	Fluctuate State
	$\max(M_a(1+d_M), M_m)$, if $\mu_M < \mu_{E_{min}}$; $M_a + M_c$, if $\mu_M > \mu_{E_{max}}$	$M_n + n \times M_{na}(1 + d'_M)$
vCPUs Tuning	$V_a/2$, if $\mu_V < \mu_{VE_{min}}$; $2V_a$, if $\mu_V > \mu_{VE_{max}}$	

M_v be the total memory size of all VMs, then the occupy rate of memory in the system is

$$r_M = \frac{M_v}{(M_t - Mo)}.$$

We have another expectation value range for the system memory utilization, define it as r_{SME}. If $r_M < r_{SME_{min}}$, the minimum value of the range r_{SME}, that means we have enough free physical memory resource to distribute. We have proved that more memory is generally better. In such case, we can allocate certain amount of memory, say M_b, to each VM. If $r_M > r_{SME_{max}}$, the maximum value of the range r_{SME}, the memory usage has been reaching the upper critical point. It is not possible to allocate more memory on VMs, the only way we can do is to migrate the VMs to some relatively idle physical machines to release the stress of current VMs.

The strategy for physical CPU resource is similar. We define an expectation value range for CPU utilization r_{SCE} and let r_C be the CPU utilization we collect from /proc files. If $r_C < r_{SCE_{min}}$, the minimum value of the range r_{SCE}, tuning up vCPU number on each VM or creating more VMs is feasible; If vice versa, $r_C > r_{SCE_{max}}$, the maximum value of the range r_{SCE}, that means the server is really busy with computing. We will definitely have a very high vCPU utilization.

However, tuning up vCPU or creating more VMs will only make servers overburden at this time. One solution is to migrate some VMs onto other idle servers or to request VM instances in public cloud through the corresponding interface.

Another measure value of System Strategy is the average performance. Scheduler will calculate the average performance for the time interval when a new 1% (configurable) of work loads finishes. If it is lower than expected (a history or domain expert value), Scheduler will check the current resources throughput and propose corresponding measures to improve performance. For example, it will concentrate on CPU utilization for loosely coupled computational workloads and make it not to stay at 100% but also not very low; For tightly coupled CPU-intensive workloads, it needs to make the resources utilization in a reasonable range and allocating new VMs is necessary if utilization is already rational. In a cloud system, the administrators can get different performance and utilization results by setting a loose or tight threshold.

4.2 VM Malleability Strategy

First, VM malleability means the ability to change the ratio of VMs/PMs dynamically. VM malleability uses two procedures: VM *split* and VM *merge*. One VM can be split to multiple VMs, and subtasks are migrated to the new VMs accordingly. Multiple VMs can be merged to one VMs with the sub tasks also being merged.

Let M_l and M_h be the lowest and highest memory size on each VM respectively. Let V_l and V_h be the possible least and most vCPU number on each VM. These are all empirical values obtained from previous experiments. Memory less than M_l will cause JVM fails to run and more than M_h will influence future normal operations; while vCPU number more than V_h and less than V_l will highly influence workload's performance.

In previous sections, we have discussed the strategies for memory ballooning and vCPU tuning. VM number adjustment comes from after memory ballooning and vCPU tuning fail to reach the expectation. That means the only thing we can do is to adjust the total number of VMs. The strategy works as follows.

Let $VM_n(t)$ be the total VM number in the system at time t. If μ_M is less than $\mu_{E_{min}}$ and M_n is already getting to the critical point M_l, then we need to decrease the VM number $VM_n(t)$ to half on each VM; if vice versa, μ_M is greater than $\mu_{E_{max}}$ and M_n is already getting to the critical point M_h, we will double the VM number. While for vCPUs, if μ_V is less than $\mu_{VE_{min}}$ and V_n is already getting to the critical point V_l, then we need to decrease the VM number VM_n to half on each VM; if vice versa, μ_V is greater than $\mu_{VE_{max}}$ and V_n is already getting to the critical point V_h, we will double the VM number.

To do VM malleability, the scheduler will check the output of the VM malleability strategy when every 1% job has been done and send out VM splitting or merging instruction (based on the strategy results) to VM number controller. Meanwhile the scheduler will stop running the workload temporally and migrate corresponding actors forward to new VMs or back to the same VM, and then resume the execution. This modulation is also iterative. The Scheduler will check memory and CPU utilization under the new configuration and take further adjustment, either memory ballooning or vCPU tuning or VM number change based on corresponding memory and CPU utilization.

In summary, the new number of VMs $VM_n(t+1)$ is defined as:

$$VM_n(t+1) = \begin{cases} VM_n(t)/2, \text{if} \mu_M < \mu_{E_{min}} \wedge M_n \cong M_l \\ VM_n(t) \times 2, \text{if} \mu_M > \mu_{E_{max}} \wedge M_n \cong M_h \end{cases} \tag{4.1}$$

if the virtual memory size is the most important factor for performance.

Or

$$VM_n(t+1) = \begin{cases} VM_n(t)/2, \text{if} \mu_V < \mu_{VE_{min}} \wedge V_n \cong V_l \\ VM_n(t) \times 2, \text{if} \mu_V > \mu_{VE_{max}} \wedge V_n \cong V_h \end{cases} \tag{4.2}$$

if the number of vCPUs per VM is the most important factor for performance.

4.3 Memory Ballooning Strategies

We distinguish the cloud computing environment into stable condition and the fluctuant situation. Here fluctuation indicates that new workloads come into the system or existing workloads finish. The framework has to rearrange the resource allocation in order to get to a temporary steady state. Under stable condition, the strategy aims to bring all workloads to the ideal or optimum state.

Before entering the main body, let us define the new adjusted memory on each VM as $M_a(t+1)$, a function of time t. Next we will discuss the corresponding strategies in two different states.

4.3.1 Stable Condition

First we will discuss about memory ballooning in stable conditions, which means the amount of workloads in the environment are immutable.

Let $M_a(t)$ be the memory allocated at time t and M_u be the average usage for current workload at some time, then

$$\mu_M = \frac{M_u}{M_a(t)}$$

is the VM's average memory utilization during the time interval. There is an expectation value range for the memory utilization, define it as μ_E. After calculating μ_M, the Scheduler will check whether it is greater or less than the expectation value μ_E. if μ_M is less than $\mu_{E_{min}}$, the minimum value of the range μ_E, that means now we have low memory utilization and we need to shrink the memory on each VM. Here we have another important parameter, the allowed deviation of needed memory, d_M. It is used to guarantee every VM has surplus memory to prevent out-of-memory exception. Theoretically, the new memory size after shrinking would be

$$M_a(t+1) = M_a(t) \times (1 + d_M).$$

We also need to take the history memory usage values on each VM into consideration. Let M_m be the maximum values of desired memory among all VMs in the past, the actual new memory on each VM would be

$M_a(t+1) = M_a(t) \times (1 + d_M)$ if it is greater than M_m or $M_a(t+1) = M_m$ if it is less than M_m.

Next, we have to deal with the situation when μ_M is greater than $\mu_{E_{max}}$, the maximum value of range μ_E. In such case, the memory utilization is kind of high, we need to allocate more memory for each VM to promise the smooth future execution.

Another parameter is needed to help with adjusting the virtual memory size. M_c is an empirical constant to add upon current memory. Now the size of the virtual memory on each VM should be $M_a(t+1) = M_a(t) + M_c$.

In summary,

$$M_a(t+1) = \begin{cases} \max\{M_a(t) \times (1 + d_M), M_m\}, \text{if} \mu_M < \mu_{E_{min}} \\ M_a(t) + M_c, \text{if} \mu_M > \mu_{E_{max}} \end{cases} \tag{4.3}$$

4.3.2 Fluctuate State

Next, we will discuss the corresponding strategy in fluctuate state. When new actors (new workloads or more actors added into the same workloads) arrives, the system passes from steady state to fluctuate state. In such case, the memory on each VM should be increased to let the new actors to fulfill their tasks. Let M_{na} be the pre-defined memory size for each actor. M_{na}, which can be adjusted dynamically, is an empirical value learning from history memory usage. Now the virtual memory on each VM should be

$$M_a(t) + n \times M_{na} \times (1 + d'_M),$$

where $M_a(t)$ is the allocated memory at time t for each VM in steady state, n is the number of new actors arrive into the VM, and d'_M is the allowed deviation for new actors (generally it is larger than d_M).

4.4 vCPU Tuning Strategy

In this section, we will talk about the corresponding vCPU tuning strategies. Here we don't need to consider different environment states for vCPU tuning. First we define the new vCPU number on each VM as $V_a(t+1)$, a function of time t.

Let $V_a(t)$ be the vCPU number specified on each VM for current workload, and μ_V be the average CPU utilization during the time interval. There is an expectation value range for CPU utilization, define it as μ_{VE}. The Scheduler will check whether μ_V is greater or less than the expectation value μ_E. If μ_V is less than $\mu_{VE_{min}}$, the minimum value of the range μ_{VE}, that means now we have low CPU utilization and we need to decrease vCPU number per VM. If μ_V is greater than $\mu_{VE_{max}}$, the maximum value of the range μ_{VE}, it means we need to add more vCPUs to share the responsibility.

This adjustment is not a single-step but rather an exponential modulation process. If μ_V is less than $\mu_{VE_{min}}$, we will decrease the vCPU number to half on each VM; and double vCPUs if μ_V is greater. Next, the scheduler will check the new CPU utilization after some time interval to see the whether it satisfies our expectation. If yes, the scheduler will stop tuning vCPU for a moment; if no, the scheduler has to continue tuning up or down the vCPU number by following the "half" or "double" rule. One prerequisite is that the vCPU number should not be less than physical CPUs on all VMs.

In summary,

$$V_a(t+1) = \begin{cases} \frac{V_a(t)}{2}, \text{if} \mu_V < \mu_{VE_{min}} \\ V_a(t) \times 2, \text{if} \mu_V > \mu_{VE_{max}} \end{cases} \tag{4.4}$$

The resource usage monitor continues to check μ_V.

5. Performance Evaluation

In this chapter, we will show how the threshold based strategy works to refine the virtual network configurations. First we will apply this strategy to some simple cases, for example, vCPUs tuning, memory ballooning for VMs and the mixed cases. Next, we will test the strategy on VM merging and splitting and analyze whether we can benefit from the adjustment. Last, we will apply the strategy on some more complex cases, which may integrate vCPU tuning, memory ballooning, VM merging or splitting, and VM migration together, sometimes even adjust these configurations repeatedly. We will compare the cost and benefit by adopting this strategy into a virtual system at workloads' runtime.

5.1 Evaluation of Simple Scenario

In the simple scenario, we will deploy the threshold based strategy on dynamically VM configuration changing, which includes changing the number of vCPUs, the size of virtual memory on each VM.

5.1.1 Evaluation of vCPUs Tuning

The workloads for evaluating vCPUs tuning is Stars Distance Computing. In the experiment, we set the number of VMs as 4 and the memory size of each VM as 1024MB. Table 5.1 shows the performance of different vCPUs per VM.

The utilization of CPU is very high (100% for most tim) when we have only 1vCPU per VM. The Scheduler receives the CPU consumption from Resource Usage Monitor and finds the average utilization is greater than the threshold; therefore, vCPUs tuning instruction is passed to vCPU Controller. We can see from Table 5.1 that the new running time is shorter than 1vCPU per VM configuration but still longer than 4vCPU one.

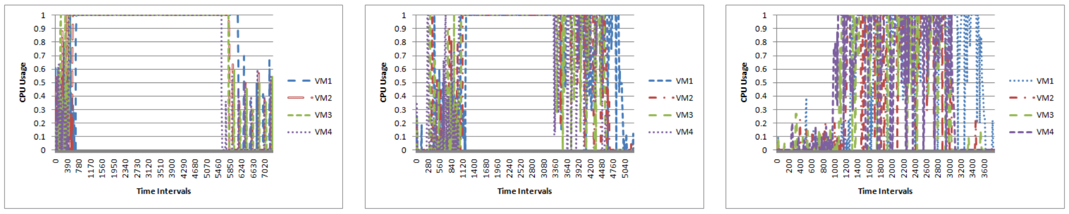
Table 5.1: Evaluation of threshold-based strategy on vCPUs tuning

the number of vCPUs per VM	1vCPU	2vCPUs	1vCPU→2vCPUs
running time	65.9s	37.5s	50s

Table 5.2: Evaluation of threshold-based strategy on memory ballooning

the memory size per VM	512MB	1024MB	512MB→1024MB
running time	29.6s	22.5s	24.1s

Figure 5.1 shows the corresponding CPU usage on 1vCPU, 2vCPUs and automatic vCPUs tuning VM configurations. We can see that the CPU usage with automatic tuning is between the 1vCPU and 2vCPUs settings. It accords with the performance changing after vCPU tuning.



(a) 1vCPU (b) mix vCPUs (c) 4vCPUs

Figure 5.1: CPU usage on different vCPU settings

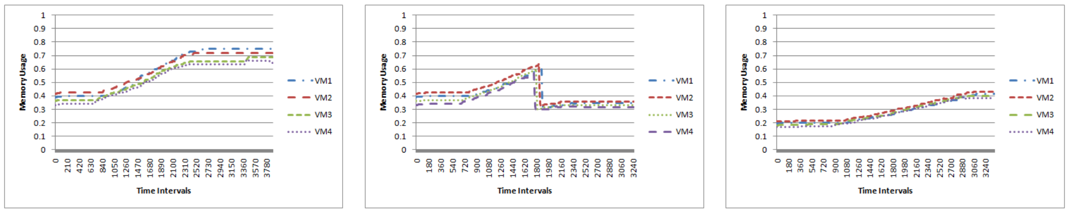
If we allocate a high number of vCPUs per VM, the Scheduler will send a vCPU tuning down instruction to vCPU Controller because of low utilization (based on the threshold we set in Scheduler). However, this adjustment will not bring any performance improvement for loosely coupled workloads. The administrator can avoid this kind of adjustment by changing the CPU utilization threshold. If the workload is tightly coupled, for example Heat Distribution, we will get a better performance from better CPU utilization by tuning up or down the number of vCPU per VM.

5.1.2 Evaluation of Memory Ballooning

The workloads for evaluating memory ballooning Heat Distribution. In this experiment, we set the number of VMs as 4 and the number of vCPUs of each VM as 4. Table 5.2 shows the performance of different memory settings.

The utilization of memory is higher than our expectation (reasonable memory usage threshold) for 512MB per VM setting. The Scheduler receives the memory consumption from Resource Usage Monitor and checks available memory of the under PMs. If the PMs have enough memory to allocate, the Scheduler passes memory ballooning instruction to Virtual Memory Controller to balloon up virtual memory for each VM. We can see from Table 5.2 that the new running time is shorter than 512MB memory configuration but still longer than 1024MB one.

Figure 5.2 shows the corresponding memory usage on 4VMs for 512MB, 1024M-B and automatic memory ballooning VM configurations. We can see that the memory usage with automatic tuning is increasing at first then decreasing at the point around 50%. The turning point happens when the virtual memory changes from 512MB to 1024MB for each VM.



(a) 512MB (b) mix memory (c) 1024MB

Figure 5.2: Memory usage on different memory settings

If the memory allocated to each VM is very large and thus have a very low utilization, the Scheduler will send a memory ballooning instruction to Virtual Memory Controller to reduce the memory size. The memory usage will first grow slowly then increase sharply to a higher point and then grow slowly again. The first turning point happens when virtual memory changes from a larger memory to a smaller one. For tightly coupled CPU-intensive workloads, the performance will change following the memory changes; the performance has no significant change for loosely couple computational workloads. Therefore, the administrators can set a loose threshold for loosely coupled workloads but set a tight threshold for tightly coupled workloads. In a summary, the administrator can get what they expect, for example better performance, better utilization or a balance between these two by setting different threshold in the Scheduler.

5.1.3 Evaluation of VM Malleability

To evaluate the strategy on VM malleability, we run both Stars and Heat workloads in our framework. In this experiment, Table 5.3 shows the performance of different memory settings.

We set the number of VMs as 2, memory size as 512MB and number of vCPUs as 4 for initial state of Heat workload. The performance under such virtual network configurations is beyond our acceptable value range. Yet increasing memory size or number of vCPUs will not improve performance significantly. The Scheduler first checks whether the number of VMs is still lower than the highest expectation. meanwhile it will also check whether there are enough resources in PMs by sending request to Resource Usage Monitor. If reply is yes, Scheduler then sends a VM splitting instruction to VM number Controller.

Next we set the number of VMs as 16, memory size as 1024MB and number of vCPUs as 4 for initial state of Heat workload. The performance is bad under such configuration and the resource usage is lower than the threshold we set. Decreasing memory size or number of vCPUs per VM is not reasonable according to the history data, it is better to merge the VMs.

For loosely coupled computational workloads, the purpose of doing VM malleability is for energy saving or resources relief. The resource consumption does not have a severe change for tightly coupled workloads, we have to set a very harsh threshold in order to stimulate Scheduler. Table 5.3 shows that the running time after splitting or merging is a little bit longer than the static configurations.

A noticeable thing here is: creating new VMs and migrating actors, which are expensive than vCPU tuning and memory ballooning alone or together, will affect the performance in some extent.

The evaluation results tell us that the threshold based strategy really works. VM malleability is feasible here, especially for loosely coupled workloads though the performance will be effected a little bit. Both private cloud owners and service providers can benefit from this framework and strategy. Private cloud owners do not need to spend a lot of resources and time on monitoring execution; they can control the performance and resource utilization by adjusting threshold. Service providers

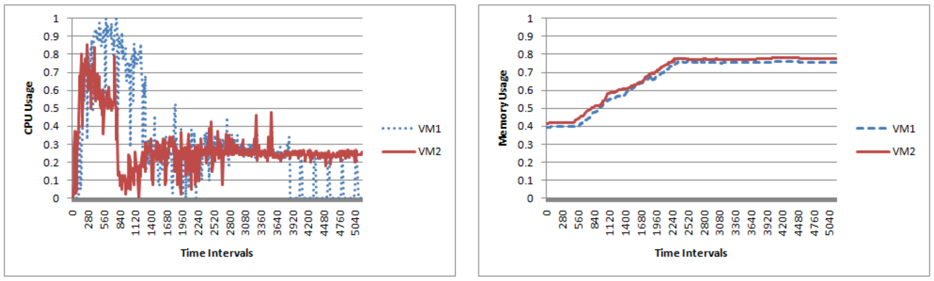
Table 5.3: Evaluation of threshold-based strategy on number of VMs

Virtual Network Configurations	Running Time (Heat)	Running Time (Stars)
2VMs	114.8s	15.3s
2VMs→4VMs	53.4s	16.6s
4VMs	15.4s	15.5s
16VMs	27.8s	14.9s
16VMs→8VMs	24.0s	16.3s
8VMs	16.5s	15.4s

can benefit from consolidating VMs properly and timely: for example, suppose we have 4VMs running for a loosely coupled workloads, consolidate another 4VMs together will have no bad influence as long as the resources in the PMs are enough.

5.2 Evaluation of Complex Scenario

In this scenario, we concentrate our evaluation experiments on Heat workload and we will see the performance and utilization change after virtual network configurations modulating. The system will finally get to some relatively ideal point through memory ballooning, vCPU tuning and VM malleability or even regulating back and forth. We set the number of VMs as 2 and each VM has 4vCPUs, the memory size of each VM is 512MB.



(a) CPU Usage (b) Memory Usage

Figure 5.3: CPU and memory usage of *Heat* on its initial state

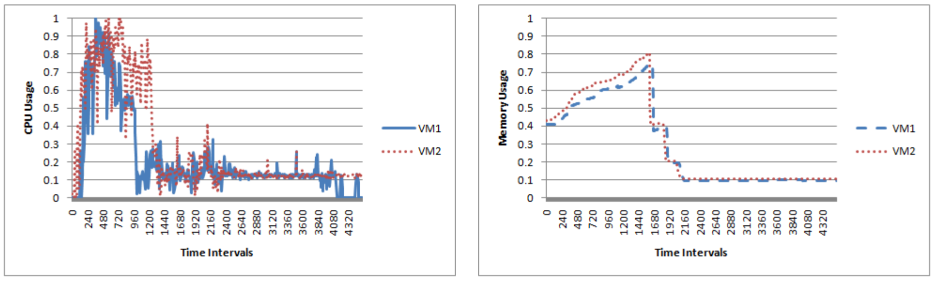
Scheduler calculates the average performance of the first 1% tasks for the initial state and finds it is in low efficiency. Thus, it asks Resource Usage Monitor for the resources utilization. Figure 5.3 shows the CPU and memory usage for the initial state. The CPU usage is high at the beginning but the slope of memory usage

Table 5.4: Evaluation of threshold-based strategy for complex scenario

Virtual Network Configurations	Running Time
2VMs with 4vCPUs and 512MB memory per VM	114.8s
Intermediate State (4→8vCPUs, 512→4096MB memory)	47.2s
2VMs with 8vCPUs and 4096MB memory per VM	18.1s
Final Stable State (2→4VMs, 8→4vCPUs, 4096→512MB memory)	29.3s
4VMs with 4vCPUs and 512MB memory per VM →8VMs	12.7s

line is steep. Thus the Scheduler will first send out memory ballooning instructions and meanwhile allocate more vCPUs on each VM to make a better throughput. The Scheduler keep trying and checking until it reaches a satisfied performance, in this scenario, the virtual network configuration is 2VMs with 8 vCPUs and 4096MB memory.

Figure 5.4 shows the CPU and memory usage when the performance gets to some reasonable point. There is no significant improvement for CPU throughput. The performance is not good enough though the new memory is enough. At this point, the Scheduler will turn to split VMs, in this example, to 4VMs first; meanwhile, the Scheduler will also decrease memory size and number of vCPUs on each VM. In this scenario, we get a perfect performance and resource utilization virtual network configuration, that is 4VMs with 512MB memory and 4vCPUs of each VM.



(a) CPU Usage (b) Memory Usage

Figure 5.4: CPU and memory usage of *Heat* on its intermediate state

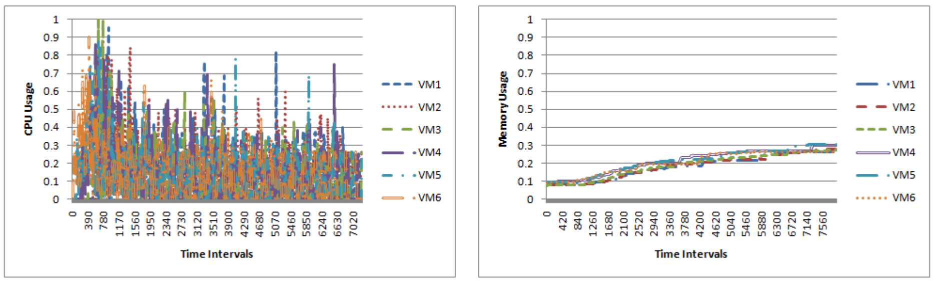
Table 5.4 summarizes the performance of different virtual network configurations for the complex scenario.

We will not show the evaluation results for VM consolidating or merging here, since it is a much easier procedure than VM splitting. When the CPU and memory utilization are both very low, on the promise of reasonable performance, the Scheduler will consolidate the actors onto some of the VMs and save others for future use or just shut down to save resources.

5.3 Evaluation with VM Migration

VM migration, by definition, is to migrate a VM from the host machine to the target machine. Therefor, we introduce another server machine into this scenario. The second machine is equipped with a quad-processor, single-core Opteron, which runs at 2.2GHz, with 64KB L1 cache and 1MB L2 cache. The RAM size is 16GB and the communication between different servers is at 10GB/sec bandwidth and 7μsec latency infiniband, and 1GB/sec bandwidth and 100μsec latency Ethernet.

In our framework, whether an when to perform VM migration is mainly decided by System Strategy, but it is still based on an overall consideration. In order to trigger the system strategy, we run another application on the first server. This application has a very long running time and meanwhile uses a lot of CPU and memory resources. We still choose Heat workload to evaluate the efficiency of VM migration but expand the size of it to 720*720 with 2000 iterations. We set the number of VMs as 6 and each VM has 4vCPUs, the memory size of each VM is 2048MB.



(a) CPU Usage (b) Memory Usage

Figure 5.5: CPU and memory usage of *Heat* on before migration

Under such virtual network configurations, the utilization of vCPUs or virtual memory are both far lower than threshold. Figure 5.5 shows the CPU and memory usage on 6 VMs for Heat workload. From the graph, we can see that the utilization

Table 5.5: Evaluation of threshold-based strategy for VM migration

without migration on same server	104.3s
with migration	83.3s
without migration on two servers	69.3s

of virtual resources is already very low and it is no need to allocate more. However, the utilization of physical processors and memory are on the contrary higher than expectation and the performance collected by system strategy is much worse compared to the history data. Integrate all data together, the system strategy decides to do VM migration and output the decision and concrete migration steps to the scheduler. The scheduler has all the resources configuration information of the servers and it is always chasing for a load balance state. Therefore, in this experiments, 2VMs will be migrated to the second server. Xen provides a command `migrate` to migrate a VM from host machine to target machine; and it also supports live migration by provide an option *-l* in this command.

Before migration, the scheduler will check the configuration of VMs, especially the memory allocated to VMs. Larger memory will cost more time to finish migration. In this experiment, the memory of each VM is 2048MB and the utilization shows in Figure 5.5 is only 30%. Thus, it is advisable to shrink the memory size.

In spite of shrinking the memory size to 1024MB, it still costs some time to migrate. Another impact factor is the communication between machines. Heat is a tightly coupled CPU-intensive workload, actors communicate frequently to switch boundary data. The communication speed between actors on different machines apparently is slower than on the same machine. In a word, VM migration brings some bad effects while improving performance.

Table 5.5 summarizes the performance before and after migration, and running on different servers from the beginning.

6. Related Work

Virtualization management, which refers to the abstraction of logical resources away from their underlying physical resources in order to improve agility, flexibility, reduce costs and thus enhance business value [23], is a critical component to accomplish effective sharing of physical resources and scalability. Barham et al. present an x86 virtual machine monitor, which calls Xen, to let multiple commodity operating systems to share the underlayer hardware and promise not influencing any performance or functionality [4]. VMware is another mature VM hypervisor in virtualization market. Recently, Eucalyptus [21] is also popular in the cloud market. Xen is open-source and famous for its almost equivalent performance as the baseline Linux system, therefore, we choose Xen as our virtual hypervisor.

Existing works on virtual network management mainly focus on VM migration [15], [9], [24], [7]. Clark et al. introduce live migration and achieves impressive performance with minimal service downtimes by carrying out the majority of migration while keeping OSes running [7]. Hansen et al. present two prototypes that allow application and the underlayer operating system migration together [9]. Sapuntzakis et al. show that efficient capsule (the state of running computer across a network, including the state in its disks, memory, CPU registers, and I/O devices) migration can improve user mobility and system management [24]. The results of our experiments in this thesis will help to get more efficient capsule thus help make a better migration. Nelson et al. present a system that implements totally transparent migrations [20]. It allows a running VM to be migrated and the migration is transparent to remote clients, applications and the underlayer operating systems. Hines et al. describe the post-copy based live virtual machine migration, which means the memory transfer will be deferred until the processor state has been sent to the target host [11]. Different from [11], Bradford et al. introduce a system that pre-copies local persistent state to the target host when VMs are still running on original host [6]. Some paper focus on live migration which supports continuous device access [12], [14].

Besides, our results are important middleware level references for cloud service providers to decide where and how to place VMs, thus to improve resources utilization. Much work has been done on workload level to improve resource utilization. Tirado et al. presents a predictive data grouping scheme to scale up and down servers dynamically, aiming to improve resource utilization and implement load balance [25]. Their forecasting model is based on access history. It works for a mature portal web site, like the music portal LastFM mentioned in this paper. Wu et al. propose resource allocation algorithms for SaaS providers to help them to minimize infrastructure cost and SLA violations [30]. The main method is to decide where and which type of VMs to be initiated. Marshall et al. proposes 'a cloud infrastructure that combines on-demand allocation of resources with opportunistic provisioning of cycles from idle cloud nodes to other processes by backfill VMs', targeting to improve the utilization of Infrastructure Clouds [19].

Some paper make contributions to resource management level in virtual environments. Beloglazov et al. focus on providing more efficient VM consolidation to optimize resource usage [5]. Based on adaptive utilization threshold, they propose a method to consolidate VMs dynamically. Jin Heo et al. use feedback control to implement dynamic memory allocation in consolidated environment and they also take Xen virtual machines for research and experiments [10]. Sandpiper, presented in paper [29], successfully automates the hotspots monitoring, detecting tasks; the system implements both black-box and gray-box approaches to resolve the server hotspots problem. Zhao et al. present MEB (memory balancer) to successfully predict the memory needed for a virtual machine and reallocate memory according to the prediction. MEM will help improve the overall throughput of system, thus help improve resource utilization [31].

As mentioned before, the framework in this thesis could help private cloud owners and service providers to save cost and energy. Usually, people also care about the power saving for a system. For example, Petrucci et al. introduce and evaluates an approach for power management in a cluster system [22].

7. Discussion

7.1 Conclusion

In this thesis, we study three different categories of workloads on various virtual network configuration strategies (*i.e.*, number of VMs, number of vCPUs per VM, memory size of each VM). For the loosely-couple CPU-intensive workloads, VM configurations changing has no significant influence on performance as long as one prerequisite is met; that is the total number of vCPUs should not be less than that total number of physical CPUs. While for the tightly coupled CPU-intensive workload, like Heat, we got a more subtle relationship between VM configurations and workloads' performance. There exists an optimal configuration to get the best performance; generally larger memory brings better performance but it is worth to notice that double or even triple memory size brings around 10% improvement on performance. Virtual network configuration is not critical for performance of external network-intensive workloads.

We also collect the resource usage information for the two workloads. These data further help us to understand the impact of different virtual network configurations on workloads' performance. There exists a configuration to have the best resource usage; the best resource utilization configuration gives the best performance to tightly coupled CPU-intensive workloads; for the loosely coupled computational workloads, different resource utilizations show similar performance.

Service providers, service users and private cloud owners will all benefit from our experiments. They can treat our results as virtual network configuration references to timely adjust their VM configurations or VM requests so as to make better performance, save cost or save energy.

Based on the experimental results, we built an autonomous framework and proposed a threshold based strategy to implement virtual network reconfiguration, that is to dynamically refine the virtual network configurations for a given workload. We also built several experiments to evaluate the framework and the strategy. Our results show that they work very well at improving performance and resource utilization when the initial virtual network configuration of the workloads is not ideal. The performance or resource utilization may be not as good as the new virtual configuration but it is much better than the initial configuration. The cloud administrators can balance between resource consumption and performance by setting different threshold values.

The framework and the threshold-based strategy will help private cloud owners and service providers to make a better use of their resources to improve workloads' performance and meanwhile to save cost and energy. In reality, the scale of the workloads submitted to a cloud computing environment is much larger than the benchmarks in our experiments, the difference of performance, resource consumption or cost on different virtual network configurations is significant. Therefore, it is important to know the impact of different virtual network configurations in an cloud environment for service users, service providers and private cloud owners.

7.2 Future Work

In next steps, we will choose more typical workloads to find the relationship between virtual network configurations and workloads' performance. Also, we will expand our testbed infrastructure to a large scale environment, which needs to consider unbalanced hardware configuration. Moreover, we will design more strategies to support virtual network reconfiguration and give the evaluation and comparison for these strategies. For the strategies evaluation, we will consider the scenarios of requesting new VM instances from public clouds and compare the performance improvement of doing this with cost spent on this decision.

LITERATURE CITED

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, pages 53:50–53:58, 2010.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep.*, UCB/EECS-2009-28, 2009.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP'03, pages 164–177, New York, NY, USA, 2003.

[5] A. Beloglazov and R. Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10, pages 4:1–4:6, 2010.

[6] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 169–179, 2007.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, 2005.

[8] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, 2008.

[9] Hansen, J. Gorm, and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, 2004.

[10] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Integrated*

Network Management, 2009, IM '09, IFIP/IEEE International Symposium on, pages 630–637, 2009.

[11] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, 2009.

[12] A. Kadav and M. M. Swift. Live migration of direct-access devices. *SIGOPS Oper. Syst. Rev.*, pages 43:95–43:104, 2009.

[13] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *Parallel Distributed Processing, 2009, IPDPS 2009, IEEE International Symposium on*, pages 1–12, 2009.

[14] S. Kumar and K. Schwan. Netchannel: a vmm-level mechanism for continuous, transparentdevice access during vm migration. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 31–40, 2008.

[15] P. Liu, Z. Yang, X. Song, Y. Zhou, H. Chen, and B. Zang. Heterogeneous live migration of virtual machines. In *In International Workshop on Virtualization Technology, IWVT08*, 2008.

[16] K. E. Maghraoui. *A Framework for the Dynamic Reconfiguration of Scientific Applications in Grid Environments.* PhD thesis, Rensselaer Polytechnic Institute, 2007.

[17] K. E. Maghraoui, T. Desell, B. K. Szymanski, J. D. Teresco, and C. A. Varela. Towards a middleware framework for dynamically reconfigurable scientific computing. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*, Advances in Parallel Computing, pages 14:275–14:301, 2005.

[18] K. E. Maghraoui, T. Desell, B. K. Szymanski, and C. A. Varela. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications , IJHPCA, Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, pages 20:467–20:480, 2006.

[19] P. Marshall, K. Keahey, and T. Freeman. Improving utilization of infrastructure clouds. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011*, 2011.

[20] M. Nelson, B. hong Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *In Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[21] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, 2009.

[22] V. Petrucci, E. V. Carrera, O. Loques, J. C. Leite, and D. Moss. Optimized management of power and performance for virtualized heterogeneous server clusters. In *Cluster, Cloud and Grid Computing, CCGrid 2011, 11th IEEE/ACM International Symposium on*, pages 23–32, 2011.

[23] B. Rimal, E. Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009, NCM '09, Fifth International Joint Conference on*, pages 44–51, 2009.

[24] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, pages 36:377–36:390, 2002.

[25] J. M. Tirado, D. Higuero, F. Isaila, and J. Carretero. Predictive data grouping and placement for cloud-based elastic server infrastructures. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011*, 2011.

[26] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, pages 36:20–36:34, 2001.

[27] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 254–265, 2009.

[28] P. Wang, W. Huang, and C. A. Varela. Impact of virtual machine granularity on cloud computing workloads performance. In *Workshop on Autonomic Computational Science, ACS'2010*, 2010.

[29] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, pages 53:2923–53:2938, 2009.

[30] L. Wu, S. K. Garg, and R. Buyya. SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011*, 2011.

[31] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 21–30, 2009.

APPENDIX A
Cloud Operating System v0.1: VM Malleability Code

The following chapter provides the main algorithmic codes that implement autonomous VM malleability. It contains three Classes: `MalleabilityController`, `TheaterInfo`, and `JobInfo`. (Please check online of our lab *Worldwide Computing Laboratory*)

```
public class MalleabilityController{
    private String nameServer = null;
    private Vector<TheaterInfo> theaters =
        new Vector<TheaterInfo>();
    private Vector<JobInfo> jobs =
        new Vector<JobInfo>();
    static private MalleabilityController instance =
        new MalleabilityController();
    private StringBuffer out = new StringBuffer();
    private long startTime = 0;
    private long endTime = 0;
    private int maxJobNum = 2;
    private int migration = 0;

    static public MalleabilityController getInstance()
    {
        return instance;
    }

    public void reset()
    {
        jobs.clear();
        startTime = 0;
        endTime = 0;
        out = new StringBuffer();
        for(int i = 0; i < theaters.size(); i++)
        {
            theaters.get(i).reset();
        }
    }

    public int workingJobNumber()
    {
        int ret = 0;
        for(int i = 0; i < jobs.size(); i++)
        {
            if(!jobs.get(i).isFinished()) ret++;
        }
        return ret;
    }

    public void resetSelection()
    {
        for(int i = 0; i < theaters.size(); i++)
        {
            theaters.get(i).setSelection(false);
        }
        for(int i = 0; i < jobs.size(); i++)
        {
            jobs.get(i).setSelection(false);
        }
    }

    public void selectThearter(int i)
```

```
{
    resetSelection();
    theaters.get(i).setSelection(true);
}

public Object findSelectObject()
{
    for(int i = 0; i < theaters.size(); i++)
    {
        if(theaters.get(i).isSelecttion())
            return theaters.get(i);
    }
    for(int i = 0; i < jobs.size(); i++)
    {
        if(jobs.get(i).isSelection())
            return jobs.get(i);
    }
    return this;
}

public void setTheater(Vector<String> urls)
{
    theaters.clear();
    for(int i = 0; i < urls.size(); i++)
    {
        TheaterInfo t = new TheaterInfo();
        t.setId(i);
        t.setUrl(urls.get(i));
        theaters.add(t);
    }
}
```

```
public int getMigration() {
    return migration;
}

public void setMigration(int migration) {
    this.migration = migration;
}

public int getMaxJobNum() {
    return maxJobNum;
}

public void setMaxJobNum(int maxJobNum) {
    this.maxJobNum = maxJobNum;
}

public StringBuffer getOut() {
    return out;
}

public void setOut(StringBuffer out) {
    this.out = out;
}

public String getNameServer() {
    return nameServer;
}

public void setNameServer(String nameServer) {
    this.nameServer = nameServer;
```

```
}

public Vector<TheaterInfo> getTheaters() {
    return theaters;
}

public void setTheaters(Vector<TheaterInfo> theaters) {
    this.theaters = theaters;
}

public Vector<JobInfo> getJobs() {
    return jobs;
}

public void setJobs(Vector<JobInfo> jobs) {
    this.jobs = jobs;
}

public void createJobs(int num)
{
    for(int i = 0; i < num; i++)
    {
        jobs.add(new JobInfo());
    }
}

public int getNextTheater(int jobID)
{
    int minTheater = -1;
    synchronized(this)
    {
            int min = 100000;
            for (int i = 0; i < theaters.size(); i++) {
                if (theaters.get(i).getJobs().size()
                < min) {
                    TheaterInfo t = theaters.get(i);
                    if (t.getMaxJobNum() <=
                    t.workingJobNumber())
                        continue;
                    min = t.workingJobNumber();
                    minTheater = i;
                }
            }
            if(minTheater != -1) theaters.get(minTheater).
                addJob(jobs.get(jobID));

        }
        return minTheater;
    }

    public int migrate(int jobID)
    {
        synchronized(this)
        {
            if(migration == 0) return -1;
            JobInfo job = jobs.get(jobID);
            TheaterInfo theater = theaters.
                get(job.getTheaterID());
            int currentNum = theater.workingJobNumber();
            if(currentNum <= 1) return -1;
            int min = currentNum - 1;
            if(migration == 1) min = 1;
        int minTheater = -1;
        for(int i = 0; i < theaters.size(); i++)
        {
            if(i == job.getTheaterID()) continue;
            int num = theaters.get(i).
                workingJobNumber();
            if(num < min)
            {
                min = num;
                minTheater = i;
            }
        }
        if(minTheater != -1)
        {
            theater.removeJob(job);
            theaters.get(minTheater).addJob(job);
        }
        return minTheater;
    }
}

public void startJob(int jobID)
{
    jobs.get(jobID).startJob();
}

public void endJob(int jobID)
{
    jobs.get(jobID).endJob();
}
```

```
    public void updateJob(int jobID, int percent, int total)
    {
        jobs.get(jobID).setPercent(percent);
        jobs.get(jobID).setTotal(total);
    }

}
```

The codes written in SALSA to deal with malleability:

```
 Object updateJob(int i, Integer percent)
    {
        Object o = null;
        model.updateJob(i, percent - 1, 100);
        if(percent.intValue() > 100)
        {
            model.endJob(i);
            startJob();
            finishedJob++;
            if(finishedJob == workers.length)
            {
                printResult();
            }
            return o;
        }
        else
        {
            int next = model.migrate(i);
            if(next != -1)
            {
                cals[i]<-migrate(new UAL( "rmsp://" + model.
                getTheaters().get(next).getUrl()
                + "/a" + getUrlID() ))@
                cals[i]<-doWork(percent.intValue())@
                updateJob(i, token);
            }
            else
            {
                cals[i]<-doWork(percent.intValue())@
                updateJob(i, token);
            }
        }
        return o;
    }


public class JobInfo {
    private String name = "job";
    private long startTime = 0;
    private long endTime = 0;
    private int theaterID = 0;
    private int percent = 0;
    private int historyTheater[] = new int[102];
    private long historyTime[] = new long[102];
    private double historySpeed[] = new double[102];
    private double speed = 0;
    private double totalSpeed = 0;

    private boolean selection = false;
    private boolean changeSelection = false;

    public boolean isOnJob(int job)
```

```
{
    if(job == JOBNUM) return ture;
    else return false;
}

public void Selection(boolean state)
{
    selection = state;
}

public void changeSelection(boolean state)
{
    changeSelection = state;
}

public void startJob()
{
    startTime = System.nanoTime();
    historyTheater[0] = theaterID;
    historyTime[0] = startTime;
    historySpeed[0] = 0;
}

public void endJob()
{
    endTime = System.nanoTime();
    speed = 0;
}

public boolean isFinished()
{
    return endTime != 0;
}

public int getPercent() {
    return percent;
}

public void setPercent(int percent) {
    historyTheater[percent] = theaterID;
    historyTime[percent] = System.nanoTime();
    speed = historyTime[percent] -
        historyTime[this.percent];
    speed /= percent - this.percent;
    speed /= 1e9;
    historySpeed[percent] = speed;
    totalSpeed += speed;
    this.percent = percent;
}

public boolean isSelection() {
    return selection;
}

public void setSelection(boolean s) {
    this.selection = s;
}

public double getSpeed() {
    return speed;
}
public void setSpeed(double speed) {
    this.speed = speed;
}

public long getStartTime() {
    return startTime;
}

public void setStartTime(long startTime) {
    this.startTime = startTime;
}

public long getEndTime() {
    return endTime;
}

public void setEndTime(long endTime) {
    this.endTime = endTime;
}

public int getTheaterID() {
    return theaterID;
}

public void setTheaterID(int theaterID) {
    this.theaterID = theaterID;
}
}

public class TheaterInfo {
private String url = null;
private long startTime = 0;
private long endTime = 0;
private int id = 0;
private Vector<JobInfo> jobs = new Vector<JobInfo>();
private boolean selection = false;
private boolean changeSelection = false;
private int y = 0;
private int h = 0;
private int maxJobNum = -1;

public boolean isOnTheater(int theater)
{
if(theater == THEATERNUM) return true;
else return false;
}

public void changeSelection(int x, boolean state)
{
boolean found = false;
for(int i = 0; i < jobs.size(); i++)
{
    JobInfo job = jobs.get(i);
    boolean b = job.isOnJob(x);
    job.changeSelection(state);
    if(b) found = true;
}
changeSelection = state && !found;
}

public void isSelected(int x, boolean state)
{
    boolean found = false;
    for(int i = 0; i < jobs.size(); i++)
    {
        JobInfo job = jobs.get(i);
        boolean b = job.isOnJob(x);
        job.selection(state);
        if(b) found = true;
    }
    selection = state && !found;
}

public void addJob(JobInfo job)
{
    job.setTheaterID(id);
    jobs.add(job);
}

public void removeJob(JobInfo job)
{
    for(int i = 0 ; i < jobs.size(); i++)
    {
        if(jobs.get(i) == job)
        {
            jobs.remove(i);
            return;
        }
    }
}

    public boolean isSelection() {
    return selection;
```

```
}

public void setSelection(boolean s) {
    this.selection = s;
}

public int getMaxJobNum() {
    if(maxJobNum == -1)
    {
        return MalleabilityController.getInstance().
            getMaxJobNum();
    }
    return maxJobNum;
}

public void setMaxJobNum(int maxJobNum) {
    this.maxJobNum = maxJobNum;
}

public String getUrl() {
    return url;
}

public void setUrl(String url) {
    this.url = url;
}

public long getStartTime() {
    return startTime;
}
```

```
    public void setStartTime(long startTime) {
        this.startTime = startTime;
    }

    public long getEndTime() {
        return endTime;
    }

    public void setEndTime(long endTime) {
        this.endTime = endTime;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public Vector<JobInfo> getJobs() {
        return jobs;
    }

    public void setJobs(Vector<JobInfo> jobs) {
        this.jobs = jobs;
    }
}
```