TASK OFFLOADING BETWEEN SMARTPHONES AND DISTRIBUTED COMPUTATIONAL RESOURCES

by

Shigeru Imai

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 A. Varela, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

May, 2012

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my advisor, Professor Carlos A. Varela, for his encouragement, supervision, and support on this research. His passionate and kind guidance have motivated me throughout my time at RPI. Many thanks to the former colleagues of World Wide Computing Laboratory, Wei, Ping, Qingling, and Yousaf. They have made my time at the lab very enjoyable. I also would like to thank my family, friends, and girlfriend in Japan for their encouragement and moral support. Last but not least, thanks to people at Mitsubishi Electric Corporation for giving me a chance to study at RPI and their financial support.

ABSTRACT

Smartphones have become very popular. While people enjoy various kinds of applications, some computation-intensive applications cannot be run on the smartphones since their computing power and battery life are still limited. We tackle this problem from two categories of applications. Applications in the first category are single-purpose and moderately slow (more than 10 seconds) to process on a single smartphone, but can be processed reasonably quickly by offloading a single module to a single computer (*e.g.,* a face detection application). Applications in the second category are extremely slow (a few minutes to hours) to process on a single smartphone, but their execution time can be dramatically reduced by offloading computationally heavy modules to multiple computers (*e.g.,* a face recognition application). In this category of applications, management of the server-side computation becomes more important since the intensity of computation is much stronger than in the first category. For the first category, we propose a light-weight task offloading method using runtime profiling, which predicts the total processing time based on a simple linear model and offloads a single task to a single server depending on the current performance of the network and server. Since this model's simplicity greatly reduces the profiling cost at run-time, it enables users to start using an application without pre-computing a performance profile. Using our method, the performance of face detection for an image of 1.2 Mbytes improved from 19 seconds to 4 seconds. For the second category, we present a middleware framework called the *Cloud Operating System* (*COS*) as a back-end technology for smartphones. COS implements the notion of *virtual machine (VM) malleability* to enable cloud computing applications to effectively scale up and down. Through VM malleability, virtual machines can change their granularity by using *split* and *merge* operations. We accomplish VM malleability efficiently by using application-level migration as a reconfiguration strategy. Our experiments with a tightly-coupled computation show that a completely application-agnostic automated load balancer performs almost the same as human-driven VM-level migration; however, human-driven application-level migration outperforms (by 14% in our experiments) human-driven VM-level migration. These results are promising for future fully automated cloud computing resource management systems that efficiently enable truly elastic and scalable workloads.

1. Introduction

1.1 Motivations

The current market share of smartphones is 40% of all mobile phones in the U.S. as of September 2011 and is still expected to grow [1]. Taking advantage of their connectivity to the Internet and improving hardware capabilities, people use smartphones not only for making phone calls, but also browsing the web, checking emails, playing games, taking videos/photos, and so on.

However, some computation intensive applications cannot be run on the smartphones since their computing power and battery life are still limited for such resource demanding applications as video encoding/decoding, image detection/recognition, 3D graphics rendering, and so on. For example, a face detection application on iPhone 3GS takes 11 seconds to detect a face from a given picture (See Section 2.2.1 for detail) and people might feel it is too slow. To improve the user experience of such computation intensive applications, offloading tasks to faster computers is a common technique in the mobile computing.

We tackle this problem from two categories of computation-intensive applications. Applications in the first category are single-purpose and moderately slow (more than 10 seconds) to process on a single smartphone, but can be processed reasonably quickly when offloading single module to a single fast computer. A face detection application is an example application in this category. Applications in the second category are extremely slow (a few minutes to hours) to process on a single smartphone, but their execution time can be dramatically reduced by offloading computationally heavy modules to multiple computers. A face recognition application, which searches against a database of images to find similar faces, belongs to this category. In this category of applications, management of the server side computation becomes important since the intensity of computation is much larger than in the first category. Considering the characteristics of the two categories, we need different approaches for each category to solve this problem efficiently.

This thesis presents solutions to each category of applications. For the first category, we propose a light-weight task offloading method using runtime profiling,

which predicts the total processing time based on a simple linear model and offloads a single task to a single server depending on the current performance of the network and server. Since this model's simplicity greatly reduces the profiling cost at run-time, it enables users to start using an application without pre-computing a performance profile.

For the second category, we present a middleware framework called *the Cloud Operating System* (*COS*) as a back-end technology for smartphones. COS implements the notion of *virtual machine (VM) malleability* to enable cloud computing application to effectively scale up and down. Through VM malleability, virtual machines can change their granularity by using *split* and *merge* operations. We accomplish VM malleability efficiently by using application-level migration as a reconfiguration strategy. To achieve these purposes, we employ SALSA [2] actor-oriented language as a basis for workload migration and load balancing. We hope COS to be operated in a large data center someday, accept SALSA actors from a lot of smartphones to solve complicated tasks, and give a good experience to users.

1.2 Structure of Thesis

The structure of this thesis is as follows: In the next chapter, we will propose the solution to the first category of applications, a light-weight task offloading method for smartphones. In Chapter 3, we will then present a middleware for autonomous Virtual Machine malleability as the solution to the second category of applications. Next, Chapter 4 will introduce related work, and finally Chapter 5 concludes this thesis with future work.

2. Light-weight Adaptive Task Offloading

In this chapter, the light-weight adaptive task offloading method is presented. This method is intended to be used for the applications that are moderately slow to process on a regular smartphone. First of all, the background of the study is shown followed by the results of a preliminary experiment. Next, the proposed distributed computation model is described. Thirdly, an instantiation of the model followed by experimental results are shown.

2.1 Background

In recent years, applications running on smartphones, such as Apple iPhone and Google Android Phones, have become very popular. Most of the released applications work fine on those platforms, but some of the applications including video encoding/decoding, image recognition, and 3D graphics rendering, could take a significant amount of time due to their computationally intensive nature. Processors on mobile devices are gradually getting faster year by year; however, without aid from special purpose hardware, they may not be fast enough for those computationally intensive applications.

To improve the user experience of such computationally intensive applications, offloading tasks to nearby servers is a common approach in mobile computing. Consequently, a lot of research has been done including [3], which introduces an offloading scheme consisting of a compiler-based tool that partitions an ordinary program into a client-server distributed program. More recent efforts utilize cloud computing technology for task offloading from mobile devices [4], [5]. Both partition an application and offload part of the application execution from mobile devices to device clones in a cloud by VM migration.

This chapter previously appeared as: S. Imai and C. A. Varela, “Light-weight adaptive task offloading from smartphones to nearby computational resources”, in *Proc. 2011 Research in Appl. Computation Symp.*, Miami, FL, 2011, pp.146-151.

On one hand, these previous studies are very flexible in terms of partitioning the program and finding the optimal execution of distributed programs; also, such methods could be applicable to a wide range of software applications. On the other hand, for simple single-purpose applications, which are typical in smartphone applications, these techniques may be too complex as partitioning, profiling, optimizing, and migrating the distributed program significantly increase running time. If an application is simple enough and a developer of the application is aware of the blocks that take most of the time, it would be reasonable to statically partition the program. Also, if application developers have done experiments on a target problem and are knowledgeable about the characteristics of the problem, they could formulate a function to predict the performance or cost of the application execution with less effort in run-time profiling.

In this paper, we describe a simple model to predict the performance of distributed programs. The model updates prediction parameters on each run to adapt to network and server speed changes in a very light-weight manner. In particular, the model does not require analyzing and profiling the application developer's original program, but it assumes the program contains a basic model that reflects the nature of the task to predict the performance. We present the design of the model as well as implementation and evaluation with a face detection problem running on an iPhone as an example. Our evaluation shows that the model predicts the actual performance very well as we iterate problem executions.

2.2 Preliminary Experiment

2.2.1 Sample Application: Face Detection

Face detection is an application of image processing that determines the locations and sizes of human faces on given digital images as shown in Figure 2.1. It is commonly used in digital cameras that detect faces to make those faces look better through image processing. Another example is a photo sharing service; such as Google Picasa or Facebook, which helps users add name tags for detected faces on photos. In those uses of face detection, the processing speed is pretty fast; users may even do not notice when

those photos are processed because they are processed on fast hardware for image processing or on servers where plenty of computational resources are available.

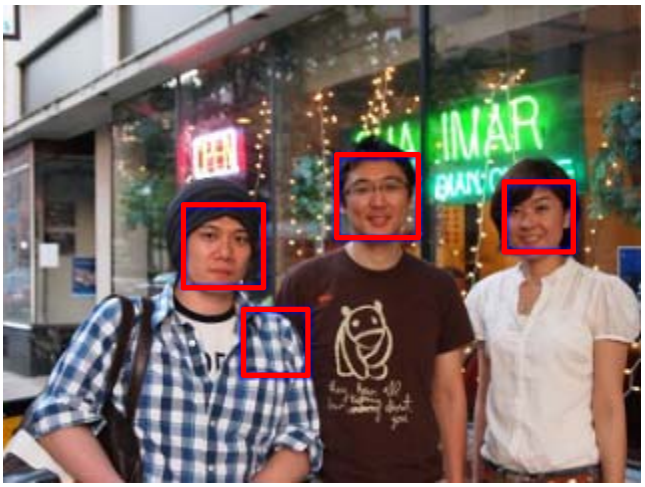

Figure 2.1: Example Result of Face Detection

However, there may be a case that users want to detect faces right after they take a photo using smartphones. One example of such usage is a make-up game application where the user can try various make-ups or hair styles with the photo without specifying a region of the face. For this kind of application, immediate face detection is critical for a good user experience.

2.2.2 Implementation of Remote Face Detection

To measure the performance of face detection on real smartphone hardware, we implement a face detection application on an Apple iPhone 3GS using the OpenCV [7] library based on an existing implementation [8] . OpenCV is a widely used open-source library for computer vision and provides general-purpose object detection functionalities.

In addition to the standalone face detection on the iPhone, the application is capable of sending an image to a server and receiving detected results. The system diagram of the application is shown in Figure 2.2.

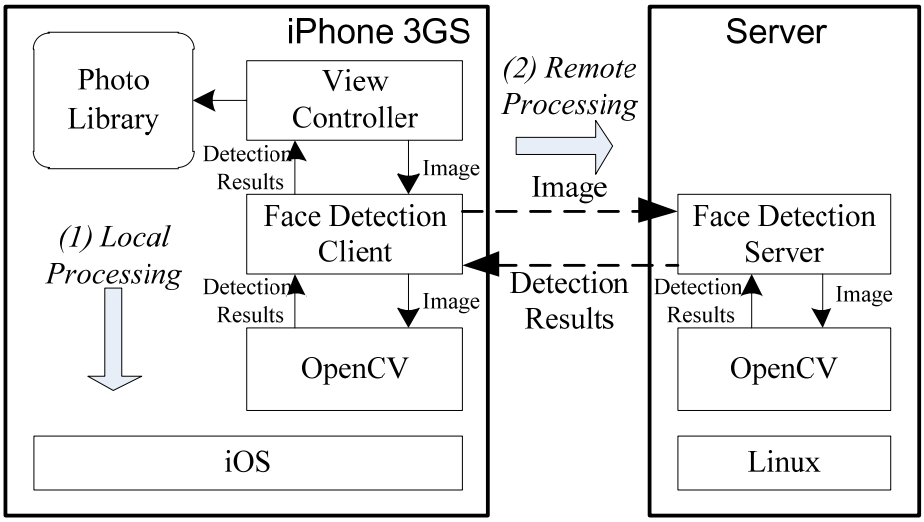

Figure 2.2: System Diagram of Application Task Offloading

The *view controller* on the iPhone first takes an image from the *photo library* specified by the user and then gives the selected image to the *face detection client*. Based on a predetermined setting, the face detection client does either (1) local processing or (2) remote processing. In the case of (1) local processing, the face detection client processes the image using the local OpenCV library and gets the face detection results. In the case of (2) remote processing, the face detection client sends a face detection request to the server over a network (*e.g.,* WiFi or 3G) connection. Once the *face detection server* receives the request, it processes the image using the OpenCV library on the server and sends the result back to the face detection client on the iPhone. Finally, the view controller receives the detection results from the face detection client and visualizes the results as shown in Figure 2.1.

2.2.3 Experimental Setup

In the experiment of face detection, we use an Apple iPhone 3GS (OS: iOS 4.2.1, detail specifications of CPU and memory have not been made public) and a server computer (CPU: Dual Core AMD Opteron Processor 870 2GHz, Memory: 32GB, OS: Linux 2.6.18.8). Both the iPhone and the server are connected in the same network segment via WiFi 802.1g. The experiment tested 33 different images whose sizes range

from 500K to 1.2 Mbytes. Those images contain at least a single face and at most eleven faces.

The version of OpenCV is 2.1.0, which is used in both the iPhone and the server side. The main function to detect faces is cvHaarDetectObjects() function provided by the OpenCV library, and the last four parameters given to the function are as follows: scale_factor = 1.1, min_neighbors = 3, flags = 0, min_size = cvSize(20,20).

During the experiment, no user applications on the iPhone other than the face detection application are launched. Bluetooth and 3G communication capabilities are turned off as well.

2.2.4 Speedup by Remote Processing

We tested the performance of both local and remote processing for the 33 images and computed speedup by the remote processing compared with the local processing. We show the result processing time in Figure 2.3 and speedup in Figure 2.4 respectively.

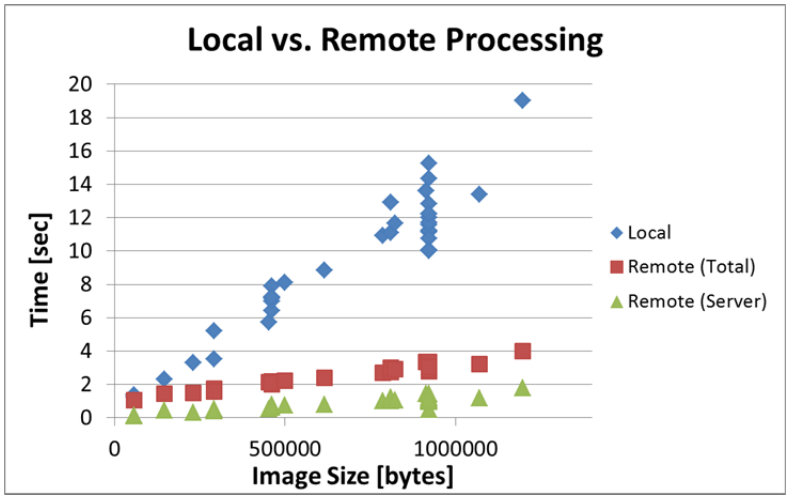

Figure 2.3: Performance of Local and Remote Processing

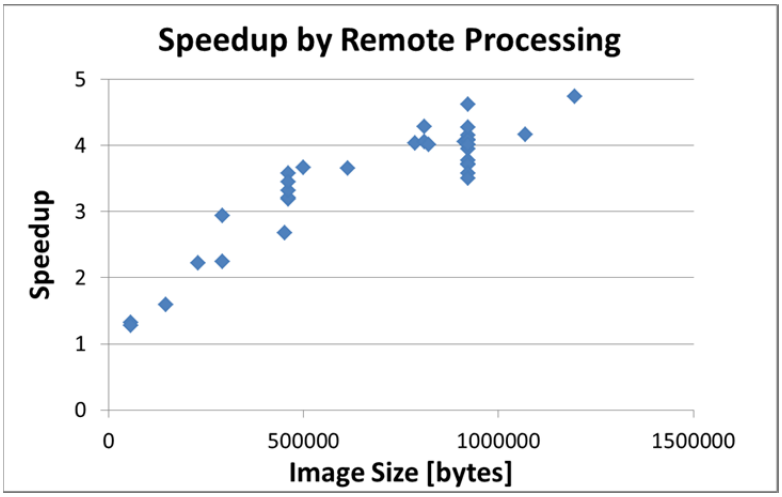

Figure 2.4: Speedup by Remote Processing

In Figure 2.3, the X-axis represents the size of each image, the plot Local represents the time required for local processing and the plot Remote (Server) represents the time required for the server to process an image when processing remotely. Remote (Total) represents the total time required for remote processing including Remote (Server) and the time for communication, i.e., sending the request from the iPhone and receiving the response from the server. From the graph, the remote processing is much faster than the local processing on the iPhone, and we can see the processing time of the images grows linearly for both local and remote processing.

Efficiency of the remote processing is clearly reflected in speedup as shown in Figure 2.4. On average, the remote processing is about 3.5 times faster than the local processing on the iPhone. An image of 1.2 Mbytes improved the performance from 19 seconds to 4 seconds, which is the biggest improvement of all the images.

The network metrics over WiFi are RTT (Round-Trip Time) = 0.0142 seconds and bandwidth = 12.8701 Mbps at the time of the experiment. The performance of the remote processing totally depends on the network environment and the speed of the server, and it works effectively if both the network and the server are fast. In the real world, there are various combinations of networks and servers, and the condition of the network changes from time to time. Therefore, the question is when to process locally or remotely. To answer this question, we need a model that predicts the performance at run-time. Similar models can be developed for battery consumption.

2.3 Distributed Computational Model

2.3.1 Remote Processing Model

We use the model depicted in Figure 2.5 to estimate total processing time when the client requests remote processing to the server. First, the user selects a problem to solve, and then the client takes it and sends a request with the problem to the server. Next, the server computes the received problem and responds a result to the client. Finally, the client shows the result to the user.

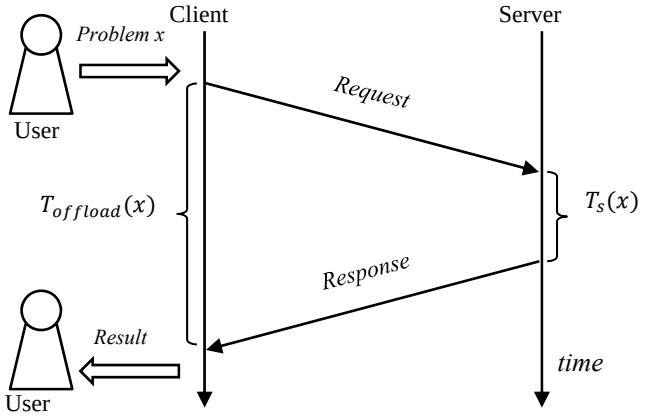

Figure 2.5: The Model for Remote Processing

Suppose the user selects a problem *x*, then the total processing time $T_{offload}(x)$ can be formulated as a function of *x* as follows:

$$T_{offload}(x) = T_{comm}(x) + T_s(x),$$

where $T_s(x)$ is the time required for the requested computation on the server and $T_{comm}(x)$ is the time required for sending the request and receiving the response.

Let $T_c(x)$ be the time required for the client to process problem *x* locally without help from the server; it is beneficial to offload the problem to the server clearly if $T_{offload}(x) < T_c(x)$.

The assumption here is that the client and the server know about basic equations of $T_{comm}(x)$, $T_s(x)$, and $T_c(x)$ for a specific problem, but do not know about the

parameters of the equations initially. This is because the client is a mobile device, and the user may use it on different networks and connect it with different servers. Moreover, because applications running on the client or the server can be downloadable on devices with various hardware settings, it is fair to assume that the applications do not know how fast the devices are until they actually run a problem on particular hardware.

2.3.2 Parameter Update Protocol

The client and the server communicate to update their internal parameters for a better prediction of computation time. To predict and compare both the local and remote processing time, we need to compute problems both locally and remotely for the first execution. From the second execution, the model predicts and compares the performance based on the parameters before running the computation, and updates the parameters accordingly from computation results. A protocol and a method to update those parameters are explained in order.

Update Protocol for $T_s(x)$ and $T_{comm}(x)$: We update $T_s(x)$ and $T_{comm}(x)$ as shown in Figure 2.6. First, the client sends a process request x' to the server, and the server processes it and measures the process time as $T_s' = T_s(x)$. Using a (x', T_s') pair and an approximation method such as least squares, the server updates $T_s(x)$ to approximate a set of measured data observed in the server. Next, the server responds with the computed result of x' back to the client as well as measured T_s' and updated $T_s(x)$. Once the client receives those parameters, then it calculates $T_{comm}{}'$ by subtracting T_s' from the measured $T_{offload}'$ by the client. Finally, the client updates its $T_{comm}(x)$ by $T_{comm}{}'$ and make a local copy from the received $T_s(x)$.

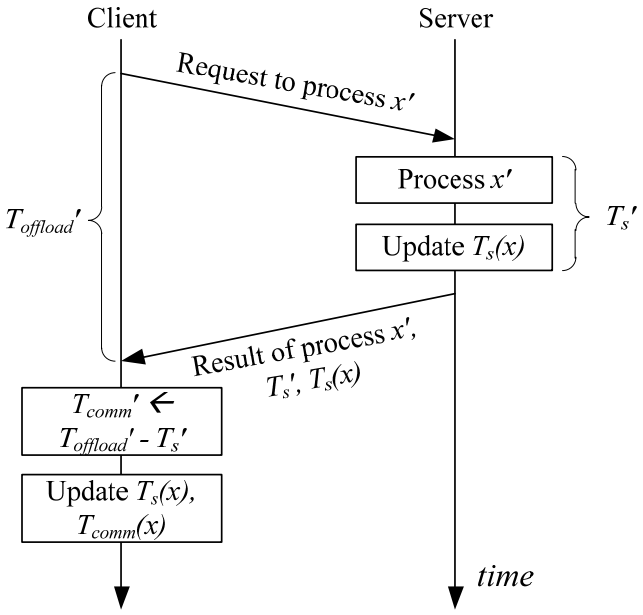

Figure 2.6: Update Protocol for $T_s(x)$ and $T_{comm}(x)$

Update Method for T_C: For the first execution of the problem or in the case $T_{offload}(x) > T_c(x)$ is true, the client locally processes the problem x'. It also measures the time required for the local process as T_c'. Just as the server updates $T_s(x)$, it updates $T_c(x)$ using a (x', T_c') pair to approximate the measured computation time. Apart from the processing time itself, the protocol and method to update parameters are light-weight so that they are able to apply at run-time without serious overhead. If a prediction error either for $T_{offload}$ or T_c becomes less than a certain threshold (*e.g.,* 10%), the above update protocols pause since it can be considered that the prediction works well enough, therefore there is no need to update the parameters. Conversely, the protocols resume if the prediction error (measured periodically with progressively less frequency) becomes greater than a certain threshold. When resuming, all the previous trained data will be reset to adapt to the new environment.

2.4 Model Instantiation

We choose the face detection as an application of the model described in the previous section. Table 2.1 shows the relationship between the parameters used in the model and their instantiation in the face detection problem.

Table 2.1: Model Instantiation

Model	*Instance*
x	An image
Processing x	Detecting face regions from an image x
$T_s(x), T_c(x)$ and $T_{comm}(x)$	Linear functions of the size of an image x
Update method for $T_s(x), T_c(x)$ and $T_{comm}(x)$	Least squares method

The reason why we choose linear functions as the models for $T_s(x)$, $T_c(x)$, and $T_{comm}(x)$ is that for this problem we know those functions to be linear as confirmed by preliminary experiments reported in Section 2.2.4. Appropriate functional models and corresponding approximation methods for updating the functions are necessary for this model to work properly.

2.5 Experimental Results

In this section, we present a detailed evaluation of our model implementation by predicting the remote processing time of the face detection task. We use the model instantiation described in Section 2.3, and we can find out how well the model predicts the remote processing time when we pick images from the 33 images used in the experiment in Section 2.2.

We perform the experiment with three different cases: case 1) The network and server are stable; case 2) The network becomes temporarily slow; and case 3) The server

becomes temporarily busy. For each case, we tested ten randomly selected sequences each consists of 33 images and measured the average time of remote processing ($T_{offload}$) and compare it with the predicted value.

2.5.1 Case 1 – Network and Server Stable

There is no dynamic disturbance from the environment in this scenario, that is, no other clients use the network and server. Figure 2.7 shows the prediction result of $T_{offload}$. As we can see from the graph, the model predicts $T_{offload}$ well. The parameters used in this prediction are generated by a relatively small number of trials as shown in Figure 2.8. In the training stage, the model generates somewhat erroneous predictions, but these prediction errors are unavoidable since there are not sufficient learning examples available at the early stage of the training. However, the least squares method works better as the trials progress. The error becomes below 10% threshold after 7.8 trials on average, and the training becomes inactive afterwards. During the inactive period of the training, the prediction error stays below 10%. The reason why the model predicted $T_{offload}$ well is that $T_{offload}$, T_{comm}, and T_s are linear functions, therefore, the least squares method works well.

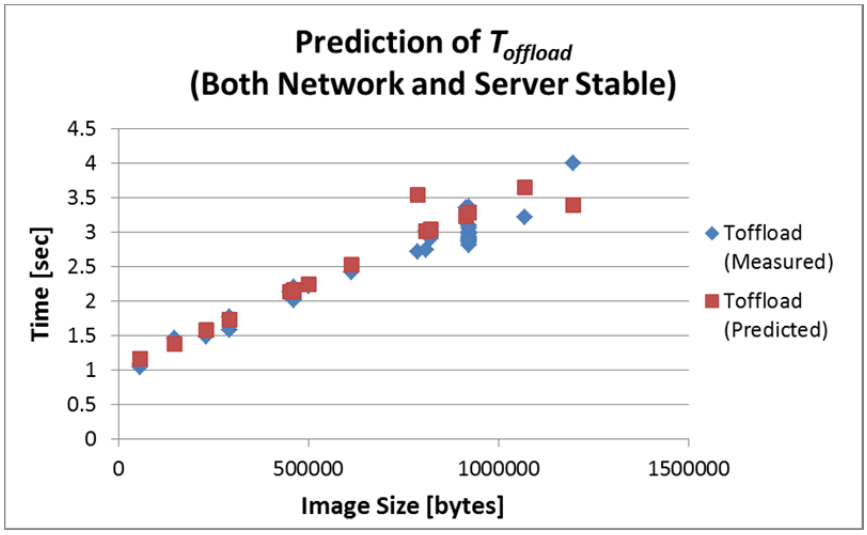

Figure 2.7: Prediction of $T_{offload}$ (Both Network and Server Stable)

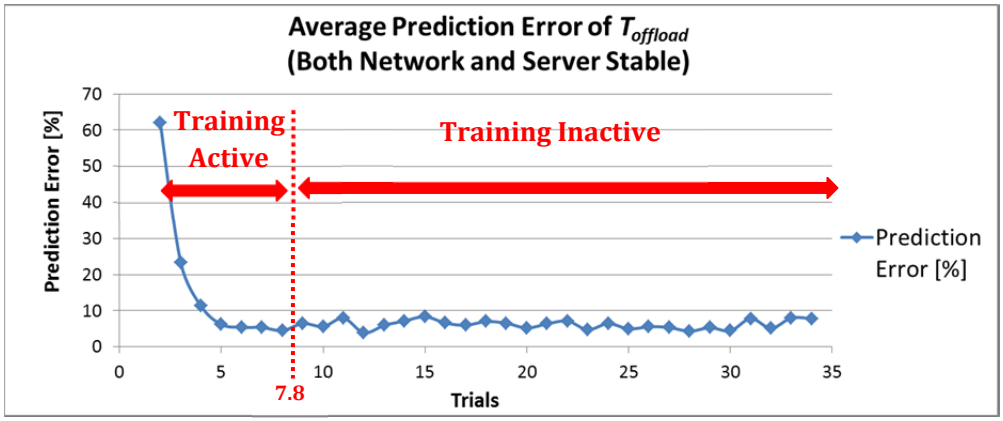

Figure 2.8: Average Prediction Error of $T_{offload}$ (Both Network and Server Stable)

2.5.2 Case 2 – Network Temporarily Slow

In this scenario, the network becomes twice slower in the middle of the trials (from the 10th to 20th) than in the rest of the trials. Figure 2.9 shows the prediction result of $T_{offload}$. The plots spread wider than Case 1 due to the disturbance, however, the model predicts $T_{offload}$ relatively well as we can see both plots for the measured and predicted are close in the graph.

In Figure 2.10, the prediction error increases quickly at the 10th trial and gradually decreases as the trial progress until it rises up again at the 21st trial. After that, the error gradually goes down again. Overall, the parameter update protocol works adaptively to the network speed change.

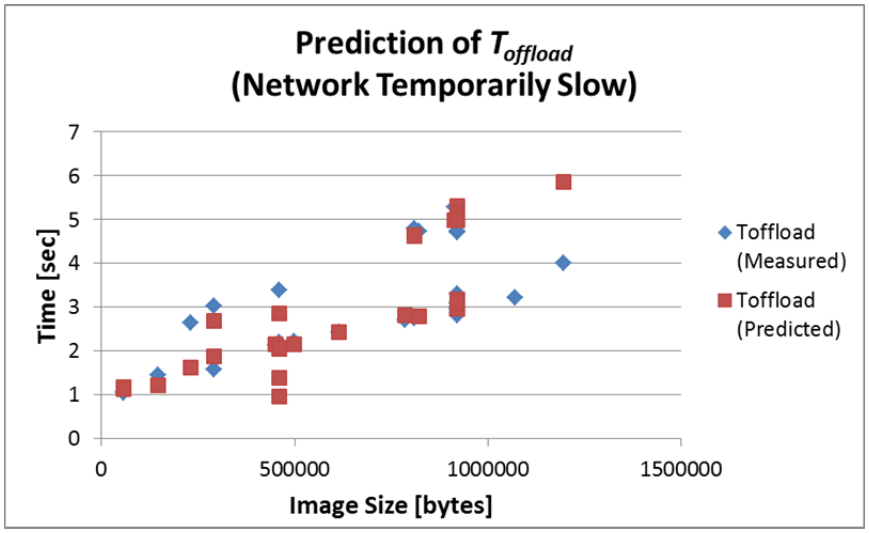

Figure 2.9: Prediction of $T_{offload}$ (Network Temporarily Slow)

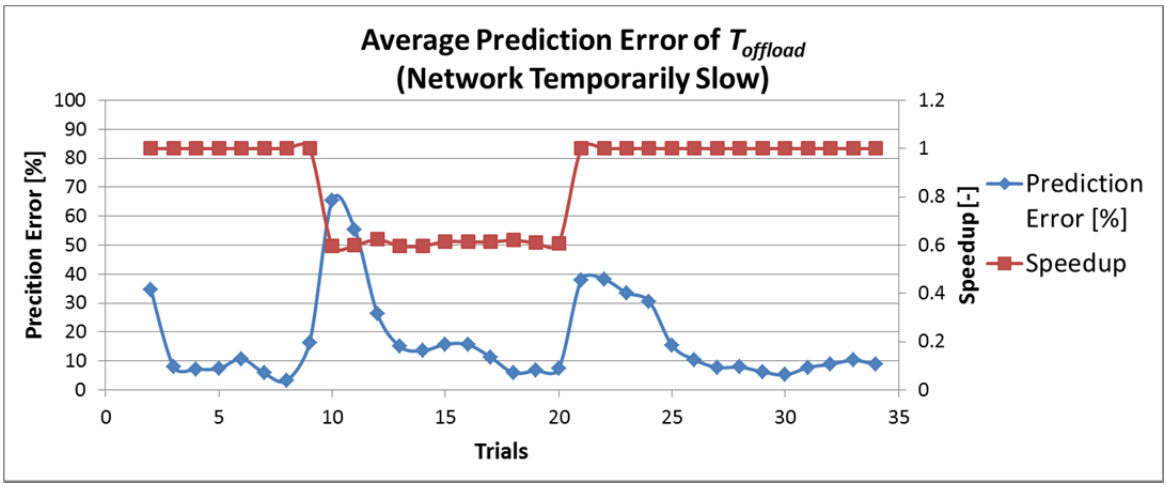

Figure 2.10: Average Prediction Error of $T_{offload}$ (Network Temporarily Slow)

2.5.3 Case 3 – Server Temporarily Busy

In this scenario, the server becomes busy and twice slower in the middle of the trials (from the 10th to 20th) than in the rest of the trials. Figure 2.11 shows the prediction result of $T_{offload}$. Same as Case 2, the plots spread slightly wider than Case 1; however, the model predicts $T_{offload}$ well despite the server overload.

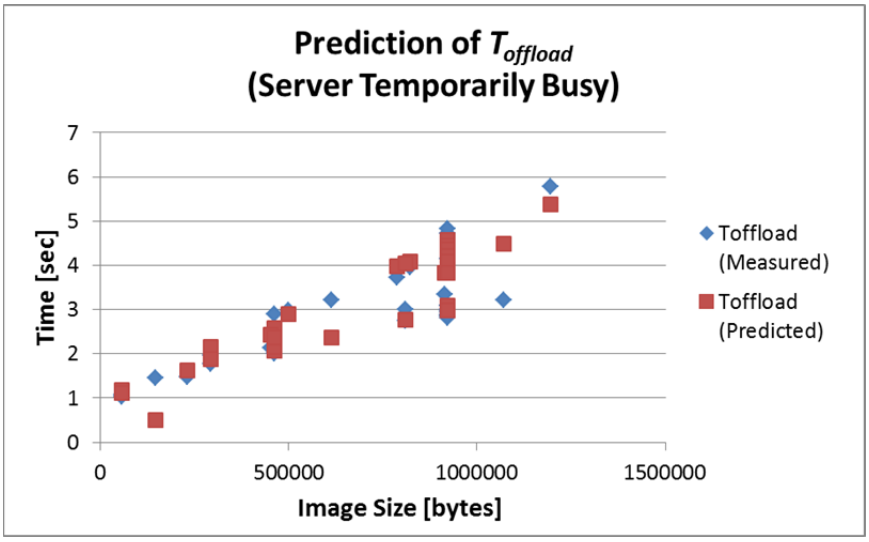

Figure 2.11: Prediction Error of $T_{offload}$ (Server Temporarily Busy)

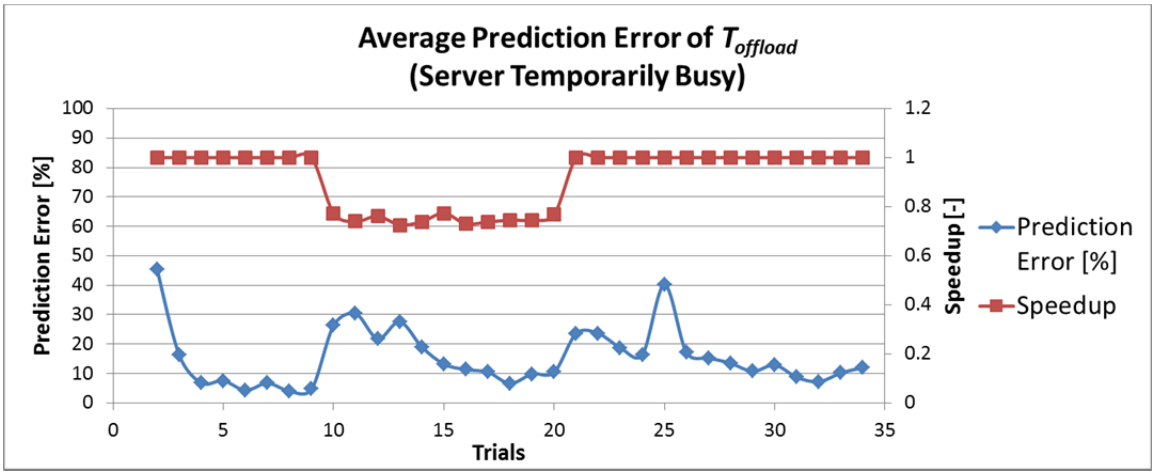

Figure 2.12: Average Prediction Error of $T_{offload}$ (Server Temporarily Busy)

Same as in the network speed change, we can see that the client can follow the server speed degradation as shown in Figure 2.12, however, the peaks of the prediction errors are lower than Case 2. This happens because the communication takes more time than processing at the server, so the effect for overall time is limited relative to the network speed change.

3. Middleware for Autonomous Virtual Machine Malleability

In this chapter, the middleware for autonomous virtual machine malleability through application-level workload migration is presented. First, the background of the study is shown followed by the result of preliminary experiments. Next, the key idea of the middleware, the concept of Virtual Machine malleability, is described. Thirdly, the proposed distributed computation middleware is introduced. Finally, experimental results are shown.

3.1 Background

Cloud computing is a promising computing paradigm that delivers computing as a service based on a pay-per-use cost model [9] . Users can purchase as many computing resources as they want paying only for what they use. This cost model gives users the great benefits of elasticity and scalability to meet dynamically changing computing needs. For example, a startup Web service company can start with a small budget, and then later if the business goes well, they can purchase more computing resources with a larger budget. Moreover, if you have a large enough budget, you would have the illusion of infinite resources available on the cloud. Those benefits have been enabled by virtualization technology such as Xen [10] or VMware [11]. Using virtualization, service providers can run multiple virtual machines (VMs) on a single physical machine, which provides isolation of multiple user workloads, better utilization of physical hardware resource enabling consolidation of workloads, and subsequent energy savings.

There are several approaches to exploit the scalability of cloud computing using VMs. Probably one of the most popular uses of cloud computing is Web services. Rightscale [12] and Amazon's Auto Scale [13] offer automated creation of VMs to handle increasing incoming requests to web servers. Conversely, when the requests become less frequent, they automatically decrease the number of VMs to reduce costs.

Another approach to support scalability is to use VM migration [14], [15], [16]. Sandpiper [17] monitors CPU, network, and memory usage, and then predicts the future usage of these resources based on the profile created from the monitored information. If a physical machine gets high resource utilization, Sandpiper tries to migrate a VM with

higher resource utilization to another physical machine with lower resource utilization and balances the load between physical machines. As a result, the migrated VM gets better resource availability and therefore succeeds to scale up its computation.

The above systems utilize VM as a unit of scaling and load balancing, and it can be done transparently for operating systems and applications on the VM. Meanwhile, the effectiveness and scalability of VM migration are limited by the granularity of the VMs. For example, the typical size of memory on a VM ranges from a few Gbytes to tens of Gbytes, and migrating such large amount of data usually takes tens of seconds to complete even within a local area network. Even though downtime by a live migration process is just as low as a few tens of milliseconds [15], it incurs a significant load on the network.

On the other hand, it is possible to achieve load balancing or scalability with finer granularity at the application workload level. This approach is advantageous since the footprint of an application workload is much smaller than that of a VM making workload migration take less time and use less network resources. Also, it can achieve better load balancing or scaling that cannot be done by the VM-level coarse granularity. However, migration or coordination of distributed execution is not transparent for application developers in general, and it requires a lot of effort to manually develop underlying software. One solution to this problem is using the actor-oriented programming model. SALSA (Simple Actor Language System and Architecture) [2] is an actor-oriented programming language that simplifies dynamic reconfiguration for mobile and distributed computing through its features such as universal naming and transparent migration. Applications composed of SALSA actors can be easily reconfigured at run time by using actor migration.

In this section of the paper, we introduce the concept of virtual machine (VM) malleability to enable cloud computing applications to effectively scale up and down. Through VM malleability, virtual machines can split into multiple VMs and multiple VMs can merge into one. We accomplish VM malleability efficiently by using application-level migration as a reconfiguration strategy based on SALSA actors. We are developing the *Cloud Operating System (COS)*, a middleware framework to support autonomous VM malleability as a PaaS (Platform as a Service) infrastructure. Actors on

COS autonomously migrate between VMs if it is beneficial in terms of resource availability in the target node, communication cost with other actors, and the cost for migration.

3.2 VM-Level vs Application-level Migration

We conduct two preliminary experiments to investigate the characteristics of VM-level migration and application-level migration. One experiment compares migration time and the other one compares the performance and the ability of each approach to balance workloads.

3.2.1 Preliminary Experiment 1 – Migration

In this experiment, we use two machines. Each has quad core Opteron processors running at 2.27 GHz, which are connected via 1-Gbit Ethernet. We create VMs by using Xen-3.2.1 and migrate them between the two machines with the memory sizes from 0.5 Gbytes to 10 Gbytes. Figure 1 shows the result of VM migration time as a function of VM size.

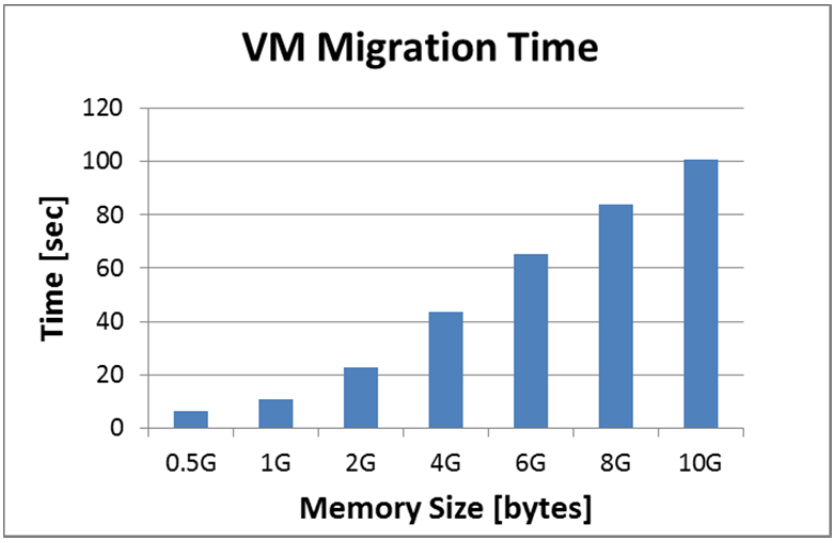

Figure 3.1: VM migration time within a local network

VM migration time linearly increases as the memory size gets larger. Roughly it takes 10 seconds per 1 Gbytes of memory, and the average bandwidth is 743.81 Mbps during the experiment. Although the size of application workloads totally depends on each application, it is expected to be much smaller than the size of VMs. In the case of

SALSA actor migration, a minimal actor (with empty state) migrates in 150 ms, whereas a 100 Kbytes actor takes 250 ms [2].

3.2.2 Preliminary Experiment 2 – Load Balancing

As illustrated in [18], the ratio between workload granularity and the number of processors has a significant impact on the performance. In the context of cloud computing, we compare the performance between the VM-based and the application-based approach.

The task is to balance the load on two physical machines (PM1 and PM2), given a workload and the number of virtual CPUs (vCPUs). The workload running on VMs is called *Stars,* analyzing three-dimensional star location data from the Sloan Digital Sky Survey. It is considered as a massively parallel CPU-intensive task. The application solving this problem is implemented in SALSA, and each actor analyzes a given set of stars data.

We test the VM-based and the actor-based load balancing with the configuration shown in Table 3.1 and Table 3.2 respectively. In the VM-based approach, VM is the unit of load balancing. In order to test against various cases of VM assignment on PM1 and PM2, we keep the number of vCPU per VM constant so that the total number of VMs changes as the number of vCPU increases. On the other hand, an actor is the unit of load balancing in the actor-based approach. Thus, we have multiple actors per one VM and assign only one VM for each PM. Note that we run the same number of actors as the number of vCPUs for both cases and assign actors on a VM evenly. For example, when we test a 12-vCPU case with 3 VMs, 4 actors are assigned on each VM.

Table 3.1: VM and vCPU configuration for VM-based load balancing

vCPU	vCPU / VM	Total VM	VM on PM1	VM on PM2
8	4	2	1	1
12	4	3	2	1
16	4	4	2	2
20	4	5	3	2

Table 3.2: VM and vCPU configuration for actor-based load balancing

vCPU	vCPU / VM	Total VM	VM on PM1	VM on PM2
8	4	2	1	1
12	6	2	2	1
16	8	2	2	2
20	10	2	3	2

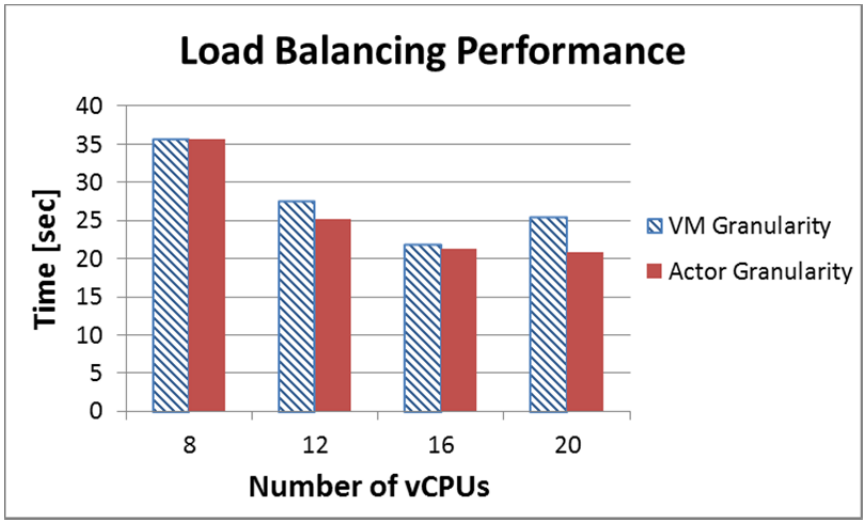

Figure 3.2: Load balancing performance

Figure 3.2 shows the result of the load balancing performance experiment. When the number of vCPUs is 12 and 20, imbalance of the number of VMs between PM1 and PM2 is unavoidable, and that degrades the performance of the VM-based load balancing. Comparing the VM-based load balancing with the actor-based load balancing, the execution time is 9.4% and 22.4% slower when the number of vCPU is 12 and 20 respectively. What happens here is illustrated in Figure 3.3 and Figure 3.4. These examples clearly illustrate that the VM-level coarse granularity has a disadvantage compared to the actor-level fine granularity in terms of load balancing.

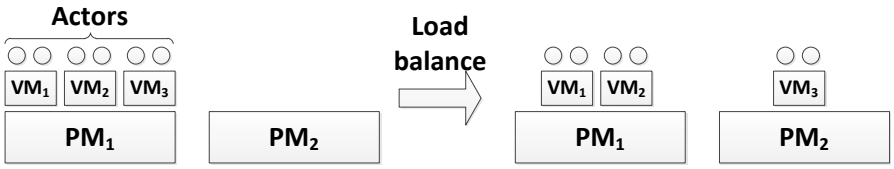

Figure 3.3: Imbalanced workload example by VM granularity

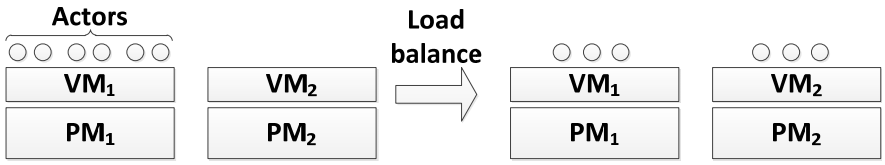

Figure 3.4: Well-balanced workload example by actor granularity

3.2.3 Design Policies towards the Cloud Operating System

Based on the observations from the previous two preliminary experiments and requirements for achieving elasticity and scalability in the context of cloud computing, we have created the following design policies towards the construction of a middleware, the Cloud Operating System:

Migration by actor granularity: As we saw in the previous experiments, application-level granularity is a key to good load balancing. Also, considering the fact that application workload migration takes significantly less time and network resources compared to VM migration, actor migration supported by SALSA is a reasonable choice since the migration can be done transparently for users. Another advantage from this approach is that we can migrate an application workload over the Internet. Xen only allows VM migration to be done within the same L2 network (to keep network connection alive), whereas SALSA application actors can migrate to any host on the Internet without service interruption thanks to its universal naming model [2]. Note that this approach works on both private and public clouds while VM migration typically is not allowed for users to do in the public cloud environment.

Separation of Concerns: Application programmers should focus on their problem of interests and leave resource management to the underlying middleware.

Autonomous runtime reconfiguration of applications: Migration of workloads should be completely transparent from the application programmer and should be done autonomously only if doing so is beneficial to the users.

Dynamic VM malleability: To fully harness the power of cloud computing, the middleware should scale up and down computing resources depending on the intensity of workloads. The middleware splits a VM as one of the available resources as needed and merge VMs into one when not needed anymore. Merging the VMs is important in terms of saving energy and costs.

3.3 Virtual Machine Malleability

We define *VM malleability* as an ability to change the granularity of VM instances dynamically. Wang et al. [19] shows that the granularity of VM makes significant impact on workload performance in cloud computing environments and first suggests the notion of VM malleability. To fine-tune the granularity of VMs transparently from users, we need to split a VM image into several parts and merge multiple VMs into fewer VMs, which are complicated tasks. Some processes have residual dependencies on a particular host and those dependencies might be broken after merging or splitting the VMs. Another issue is a resource conflict such as two http servers using port 80 migrating into one VM. We can easily imagine that realizing completely transparent VM malleability is a challenging task.

However, as shown in Figure 3.5, we realize that migrating actors to newly created VMs is equivalent to splitting a VM and migrating actors from multiple VMs to one VM is same as merging VMs. This is possible because an actor encapsulates its state and does not share memory with other actors. In a word, they are natively designed to be location-agnostic, and therefore they do not have any residual dependencies and resource conflicts when moving around computing nodes. COS implements VM malleability by migrating actors over dynamically created VMs.

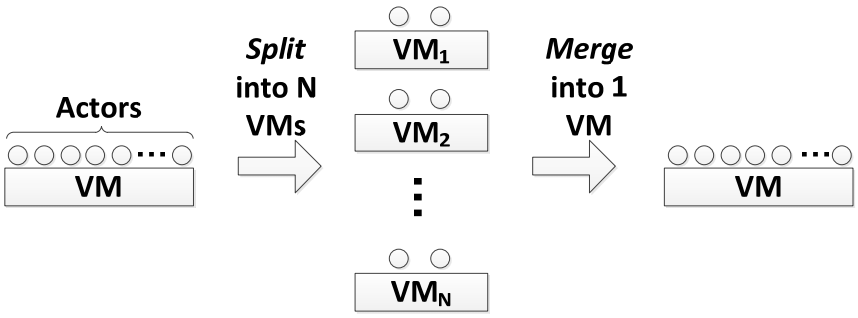

Figure 3.5: VM Malleability by actor migration

3.4 Distributed Computation Middleware

In this section, we first introduce the SALSA programming language as a key ingredient of our distributed computation middleware, and then explain about the distributed computation middleware, the Cloud Operating System.

3.4.1 SALSA

SALSA is a general-purpose actor-oriented programming language. It is especially designed to facilitate the development of dynamically reconfigurable distributed applications. SALSA supports the actor model's unbounded concurrency, asynchronous message passing, and state encapsulation [20]. In addition to these features, it offers a universal naming model with support for actor migration and location-transparent message sending.

The name service, part of the SALSA runtime, binds a Universal Actor Name (UAN) with a Universal Actor Locator (UAL) and keeps track of its location, so that the actor can be uniquely identified by its UAN. Figure 3.6 is a sample code in SALSA, which migrates an actor from host A to host B. In this example, an actor is named "uan://nameserver/migrate" and is bound to the location "rmsp://host_a/migrate" first, and then is migrate to the new location "rmsp://host_b/migrate". For more detail of SALSA, refer to the paper [2].

```
behavior Migrate {

  void print() {
    standardOutput<-println( "Migrate actor just migrated here." );
  }

  void act( String[] args ) {
      UAN uan = new UAN( “uan://nameserver/migrate” );
      UAL ual = new UAL( “rmsp://host_a/migrate” );

      Migrate  migrateActor = new Migrate() at ( uan, ual );

      migrateActor<-print() @
      migrateActor<-migrate( “rmsp://host_b/migrate” ) @
      migrateActor<-print();
  }

}
```

Figure 3.6: Migration code sample in SALSA

As illustrated in the above example, this high-level migration support in SALSA greatly reduces the development cost of COS. Therefore, we use SALSA as the basis of our middleware and also assume that applications are written in SALSA.

3.4.2 Cloud Operating System (COS)

In this section, the system architecture, runtime behavior, and load balancing algorithm of the Cloud Operating System are presented.

3.4.2.1 System Architecture

COS is based on the Xen hypervisor and we assume that Linux is used as a guest OS. It consists of several modules as shown in Figure 3.6. Each module is explained in order as follows.

- ***Actors***: Application actors implemented in the SALSA programming language. They carry workloads inside and execute them on any SALSA runtime environment.

- ***SALSA Runtime:*** The running environment for SALSA actors. It provides necessary services for actors such as migration, execution of messages, message reception and transmission from/to other actors, standard output and input, and so on.

- ***Load Balancer***: A Load balancer monitors available CPU resources and communication between actors running on a particular SALSA Runtime. It makes a decision whether to migrate or not from the monitored information. As a result of migration, actors communicating often tend to be collocated in the same SALSA runtime.

- ***VM Monitor***: A VM Monitor monitors the CPU utilization from /proc/stat on a linux file system on Domain-U. It notifies an event to the Node Manager on Domain-0 if the load goes above or below certain thresholds.

- ***Node Manager***: A Node Manager resides on Domain-0 and reports CPU load statistics to the COS Manager. Also, it creates and terminates VMs in response to event notifications from the VM Monitor.

- ***COS Manager***: The COS Manager analyzes reports from a Node Manager and requests a new VM creation or termination to other Node Managers based on its decision criteria. The system administrator of COS has to give available node information to the COS Manager.

Currently, we create only one SALSA Runtime per guest domain with multiple vCPUs per node. The reason is because we have confirmed that doing so performs better than creating multiple nodes each with one SALSA Runtime with fewer vCPUs per node.

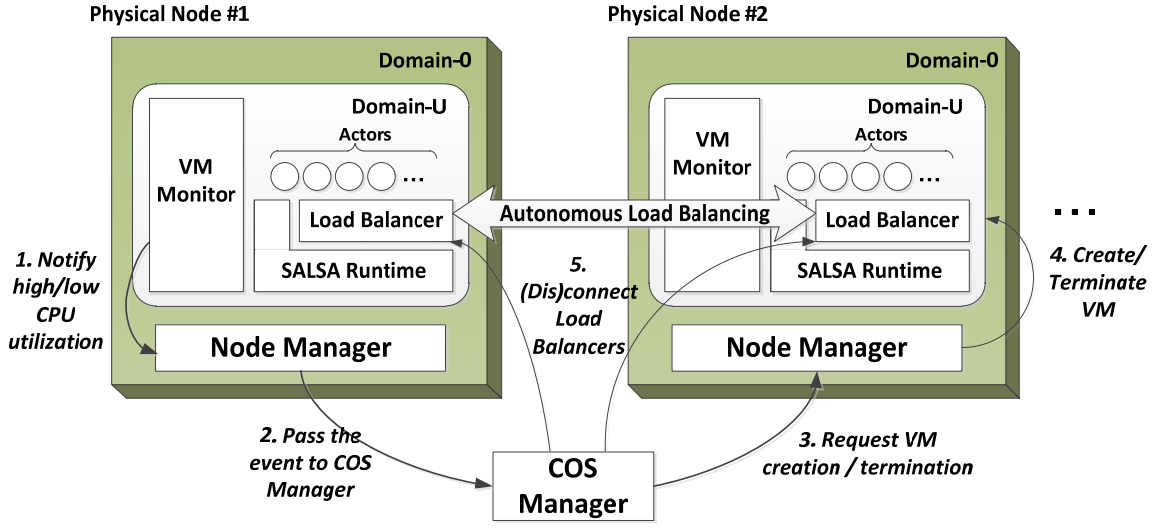

Figure 3.7: System architecture of the Cloud Operating System

3.4.2.2 Runtime Behavior of COS

Typical steps of COS when scaling up the computation are shown as follows. Similar steps apply in the case of scaling down.

Step 1. A VM Monitor in Domain-U periodically (*e.g.,* every 5 seconds) checks /proc/stat for CPU utilization to see if it exceeds certain utilization (*e.g.,* 75%). If it does, the VM Monitor then checks if more than *k* out of *n* past utilization exceeds the threshold just as Sandpiper [17] does. If it is true, it means the high CPU utilization is persistent, so the VM Monitor notifies a high CPU utilization event to the Node Manager in Domain-0 of the same physical node.

Step 2. When the Node Manager receives a high CPU utilization event, it just passes the event to the COS Manager.

Step 3. Just after the COS Manager receives a high CPU utilization event from a Node Manager, it checks if it is feasible to create a new VM. If it is, the COS Manager sends a request to another Node Manager to create a new VM with a file path to the configuration file required for Xen-based VM creation. Note that the COS Manager ignores frequent high CPU events to avoid too quick scaling up of computing resources (See next subsection for detail).

Step 4. Upon reception of a new VM creation request from the COS Manager, a Node Manager calls the '`xm create`' command to create a Xen-based new VM. After the successful creation of the VM, a VM Monitor, a SALSA Runtime, and a Load Balancer automatically start running. Once they are ready, the just launched VM Monitor sends a VM start event to the Node Manager. The Node Manager then replies a VM creation event to the COS Manager with an assigned IP address of the VM as well as the port number of the Load Balancer.

Step 5. When the COS Manager receives a VM creation event from a Node Manager, it connects available Load Balancers using given IP addresses and port information, so that the Load Balancers can start communicating with each other and balancing the load between them.

3.4.2.3 Stable Creation and Termination of VMs

To avoid thrashing behavior due to frequent VM creation and termination, the following heuristics are implemented.

- The COS manager decides to create a new VM only if it has received high CPU utilization events from all the running nodes within a certain amount of time (*e.g.* 10 seconds). The reason is that Load Balancers might be able to balance the load and decrease the CPU utilization of VMs.

- Some workload shows high and low CPU usage alternatively. If criteria of terminating VMs are too strict, this type of workload suffers very much. Let the constants to detect low and high CPU utilization be k_{LOW} and k_{HIGH} respectively for checking past n utilization. We set a higher value on k_{LOW} than k_{HIGH} to make VM termination less likely than creation.

- A VM Monitor sends a low CPU utilization event to a Node Manager only after it detects a high CPU utilization since it takes some time until a Load Balancer gets some workload from other Load Balancers.

3.4.2.4 Load Balancing Algorithm

The load balancing algorithm used in Load Balancers follows [21]. It is a pull-based algorithm where an under-utilized Load Balancer tries to 'steal' some work from other Load Balancers. This process is executed in a decentralized manner, that is, one Load Balancer passes a work-stealing request message to other Load Balancers connected locally. Thus, this algorithm shows a good scaling performance.

If a Load Balancer receives the work-stealing request message, it makes a decision whether to migrate an actor to the foreign node or not based on the following information.

- α: The amount of resource available at foreign node f
- v: The amount of resource used by actors A at local node l
- M: The estimated cost of migration of actors from l to f
- L: The average life expectancy of the actors A

The Load Balancer basically decides to migrate actors A if it observes large α and L, and small v and M. For more details, please refer to [21].

3.5 Experimental Results

We have conducted experiments from two perspectives. In the first experiment, we test COS's scalability with a minimum setting. We run COS with only one user and see how it scales up computing resources as the computation becomes intensive. Also, we compare the performance of COS with fixed amount resource cases. In the second experiment, we compare the performance of autonomous actor migration used in COS with other migration methods to figure out its advantages and disadvantages.

3.5.1 Workload

The workload used is a *Heat Diffusion* problem, a network-intensive workload. It simulates heat transfer in a two-dimensional grid in an iterative fashion. At each iteration, the temperature of a single cell is computed by averaging the temperatures of neighboring cells.

This workload is developed in SALSA, using actors to perform the distributed computation. Each actor is responsible for computing a sub-block of the grid and the actors have to communicate with each other to get the temperatures on boundary cells.

3.5.2 Experiment 1 – Test COS with Single User

3.5.2.1 Experimental Setup

In this experiment, only one user runs Heat Diffusion problem for 300 iterations on COS. We also compare the execution time of Heat Diffusion problem with the cases where the number of VMs is fixed, to see how dynamic allocation of VM affects the performance.

We use three physical machines. Each machine has quad core Opteron processors running at 2.27 GHz, and they are connected via 1-Gbit Ethernet. While COS dynamically changes the number of VMs from one to three at runtime, the fixed VM cases use one, two, and three VMs respectively from the start to the end of the Heat simulation. Note that only one VM with four vCPUs will be created per physical machine. Table 3.3 summarizes the experimental settings used in Experiment 1.

Table 3.3: VM Configuration for Experiment 1

Test Case	VM	PM	vCPU / VM	Total vCPU
Fixed VM	1	1	4	4
	2	2	4	8
	3	3	4	12
COS	dynamically allocated at runtime			

3.5.2.2 Results

VM CPU utilization: As shown in Figure 3.8, COS successfully reacts to the high CPU utilization and creates new VMs. First it starts with VM1 only, and then creates VM2 at around 21 seconds. Once both VM1 and VM2 get highly utilized, COS creates VM3 at around 92 seconds. VM malleability works well and COS successfully balances the load. The actors gradually migrate by Load Balancers, thus the CPU utilization of a newly created VM also increases gradually.

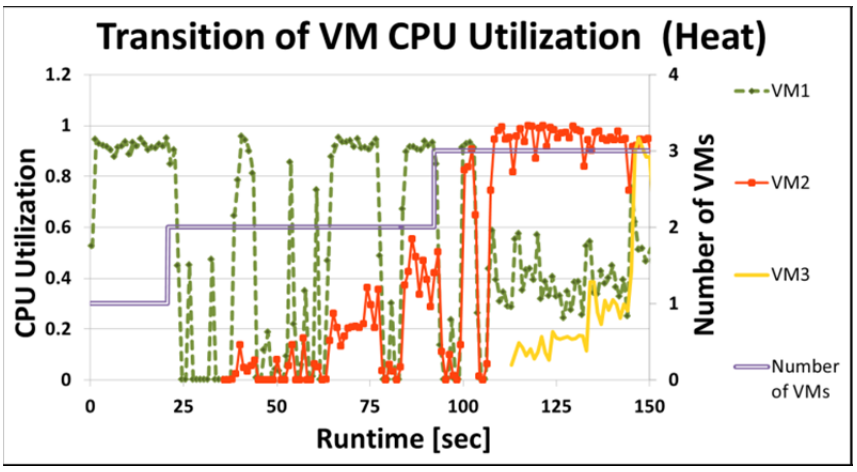

Figure 3.8: VM CPU utilization of Heat Diffusion problem running on Cloud Operating System

Throughput: Figure 3.9 depicts the throughput of Heat Diffusion simulation at each iteration. It is clear that the throughput gets higher as the number of VMs and accumulated VM CPU utilization (as shown in Figure 3.8) increases.

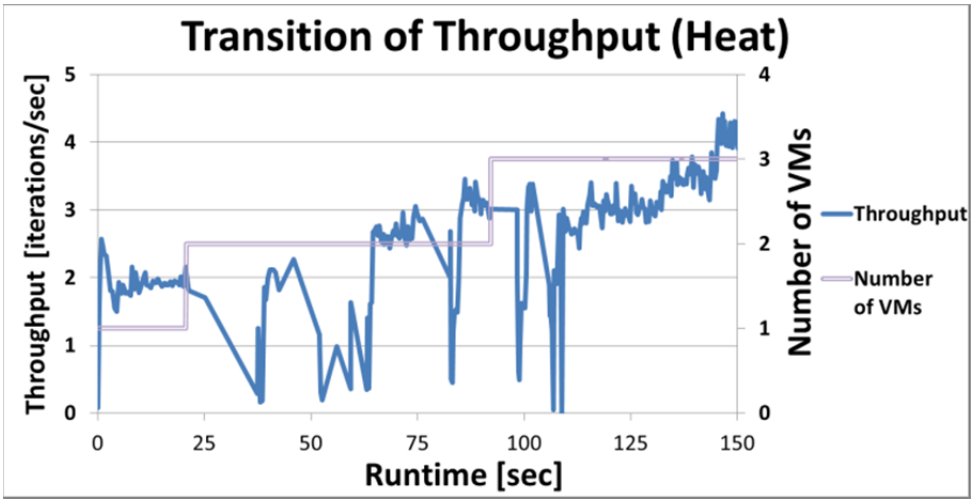

Figure 3.9: Throughput of Heat Diffusion problem running on Cloud Operating System

Execution time: Figure 3.10 shows the relationship between the number of vCPUs and the execution time. COS uses 9.002 vCPUs on average, whereas 1VM, 2VMs, and 3VMs case use 4, 8, and 12 vCPUs respectively. It is clear that more use of vCPUs makes the execution times faster. Also, the graph suggests that relationship between the number vCPUs and the execution time is linear. Suppose the cost per time is proportional to the number of vCPUs, the total cost, the product of the number of vCPUs and the execution time, is almost constant. Therefore, using 3VMs is the best way to go for the Heat Diffusion problem; however, COS does better than the cases of 1VM and 2VMs. It performs quite well if we consider the fact that COS does not have any knowledge about the complexity of the problem and the number of vCPUs is dynamically determined at run-time.

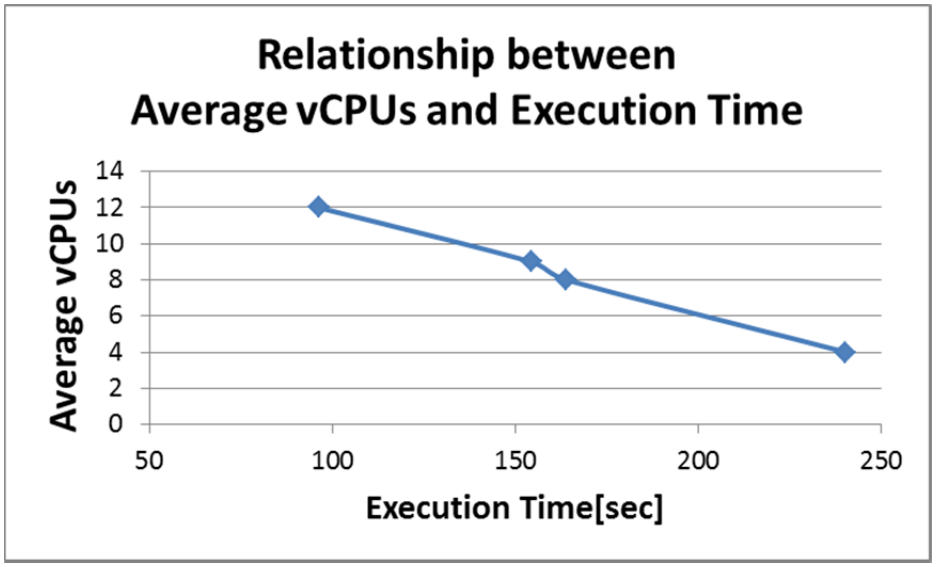

Figure 3.10: Relationship between vCPUs and Execution

3.5.3 Experiment 2 – Comparison with Other Methods with Two Users

3.5.3.1 Experimental Setup

In this experiment, we compare the performance of autonomous actor migration used in COS with other migration methods. As shown in Figure 3.11, two users submit their 8 actors each for the Heat Diffusion problem over two physical machines (PM1 and PM2). Same as Experiment 1, it runs for 300 iterations. Each machine has quad core Opteron processors running at 2.27 GHz, and they are connected via 1-Gbit Ethernet. Initially, User 1 submits his 8 actors, and then 60 seconds later, User 2 submits another 8 actors. How those actors are actually assigned on VMs depends on each test scenarios described in Table 3.4. We test autonomous actor migration along with the two cases where migrations are initiated by a human and the other two cases where no migration is triggered. Note that we used a VM that has 4 vCPUs and 1024 Mbytes of memory in each experiment.

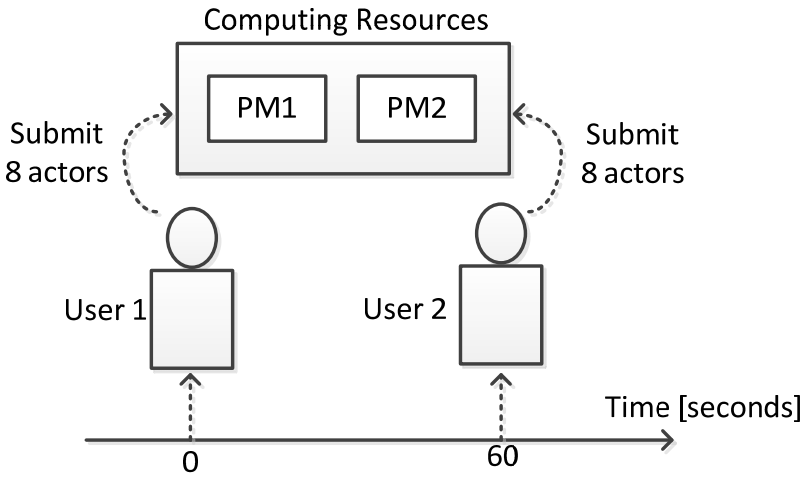

Figure 3.11: Two users submitting actors over two physical machines

Table 3.4: Test Case Scenarios for Experiment 2

Test Case Scenario	Description
VM migration (Human-driven)	- A human decides when & where to migrate VMs to achieve optimal load balancing (see next section) - User 1 has two VMs ($VM1_{U1}$ and $VM2_{U1}$) on PM1 - User 2 has two VMs ($VM1_{U2}$ and $VM2_{U2}$) on PM1 - Initially, User 1 submits 4 actors on $VM1_{U1}$ on PM1 and another 4 actors on $VM2_{U1}$ on PM1 - 60 seconds later, User 2 submits 4 actors on $VM1_{U2}$ on PM1 and another 4 actors on $VM2_{U2}$ on PM1
Actor migration (Human-driven)	- A human decides when & where to migrate actors to achieve optimal load balancing (see next section) - There is one VM on PM1 and another one on PM2 - Users do not have dedicated VMs, but share them - Initially, User 1 submits 8 actors on the VM on PM1 - 60 seconds later, User 8 actors on the VM on PM1
No migration (Fair resource)	- VMs are assigned to users in a fair resource allocation manner - User 1 has two VMs ($VM1_{U1}$ and $VM2_{U1}$) on PM1 - User 2 has two VMs ($VM1_{U2}$ and $VM2_{U2}$) on PM2 - Initially, User 1 submits 4 actors each on his two VMs on PM1 - 60 seconds later, User 2 submits 4 actors each on his two VMs on PM2
No migration (Round robin)	- VMs are assigned to users in a round robin manner - User 1 has $VM1_{U1}$ on PM1 and $VM2_{U1}$ on PM2 - User 2 has $VM1_{U2}$ on PM1 and $VM2_{U2}$ on PM2 - Initially, User 1 submits 4 actors on $VM1_{U1}$ on PM1 and another 4 actors on $VM2_{U1}$ on PM2 - 60 seconds later, User 2 submits 4 actors on $VM1_{U2}$ on PM1 and another 4 actors on $VM2_{U2}$ on PM2

Actor Migration (Autonomous)	- VM allocation and actor migration is automated - Initially, User 1 submits 8 actors on the first VM allocated on PM1 - 60 seconds later, User 2 submits another 8 actors on the first VM allocated on PM1

3.5.3.2 Human-driven Migration

In the cases of human-driven VM migration and actor migration, a human initiates migration to get the workload balanced as examples shown in Figure 3.12. Example 1 shows a case where User 1 has two VMs on PM1 and then he migrates the second VM ($VM2_{U1}$) to PM2. Example 2 illustrates a situation where User 2 newly activates his two VMs on PM1. In this case, the first VM of User 1 ($VM1_{U1}$) migrates to PM2 to collocate the VMs, which belong to the same user. Same policy applies to the human-driven actor migration.

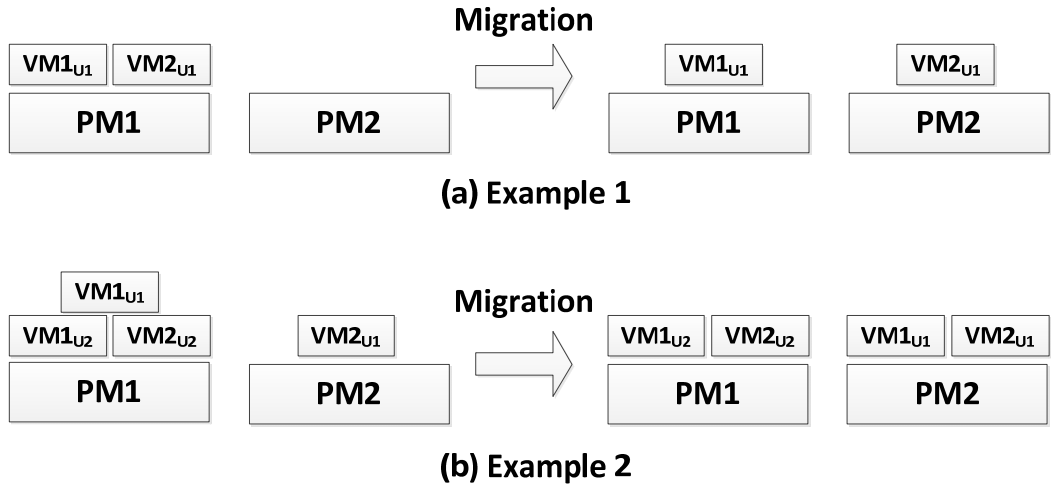

Figure 3.12: Examples of human-driven VM migration

3.5.3.3 Results

The execution time results of each test scenario are shown in Figure 3.13. We measure the execution time required for all tasks to finish. Autonomous actor migration performs almost as same as the human-driven VM migration, whereas the human-driven

actor migration is fastest of all methods and is about 14% faster than autonomous actor migration and the human-driven VM migration. This clearly shows the benefit of actor migration compared to VM migration approach. Load Balancer embedded in autonomous actor migration does not know anything about applications such as communication pattern or dependencies between modules; therefore autonomous actor migration has to profile available resources periodically and determine when to migrate actors, and that is the price autonomous actor migration is paying for the automated approach. Meanwhile, the human-driven actor migration is able to migrate multiple actors to the right place at the right time. The reason of the human-driven VM migration's moderate performance is due to the cost of VM migration, As we have shown in Section II, migration of a 1024 Mbytes memory VM takes about 10 seconds. During the migration time, communication between actors suffer significantly, therefore the performance is degraded. No migration cases are also better than the human-driven VM migration and autonomous actor migration. Obviously, there are situations where physical machines are underutilized in these two cases; however, the disadvantages of underutilization are smaller than those of the human-driven migration and the automated migration used in autonomous actor migration. In realistic cloud, with thousands of users, VMs, and workload, an automated approach is mandatory.

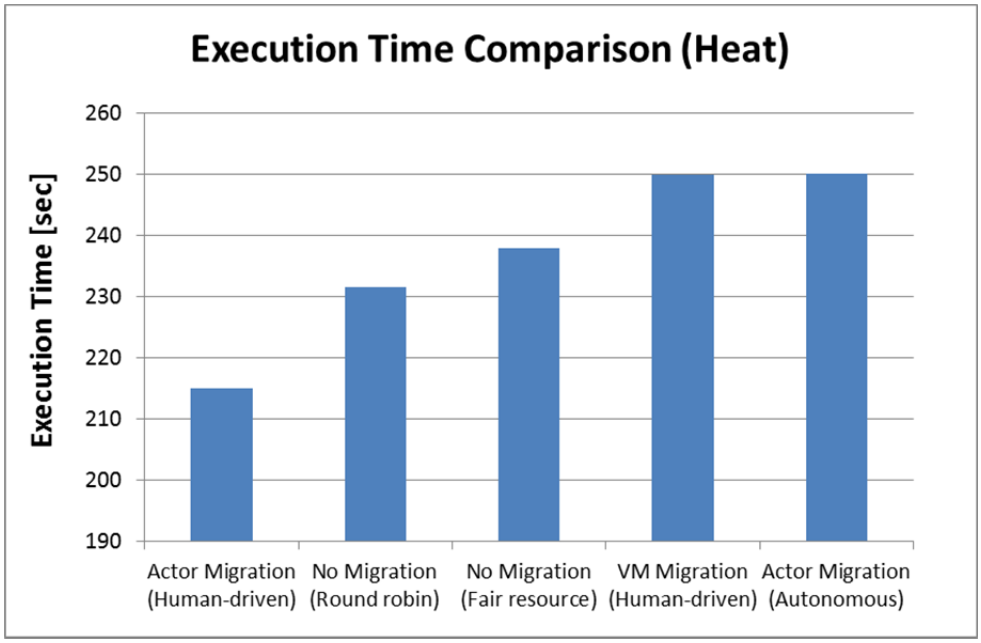

Figure 3.13: Execution time comparison of Heat Diffusion problem

4. Related Work

4.1 Mobile Distributed Computing

Wang and Li [3] introduce a compiler-based tool that partitions a regular program into a client-server program. MAUI **Error! Reference source not found.** lets developers specify which methods of a program can be offloaded for remote execution, focusing on minimizing battery consumption.

Cyber foraging [22] uses nearby surrogate computers to offload the computational burden from mobile devices. Similarly, Slingshot **Error! Reference source not found.** replicates a remote application state on nearby surrogate computers co-located with wireless access points to improve performance. These approaches are similar to ours in a sense that they assume applications are partitioned before execution and utilize nearby computers, but they do not have an explicit mechanism to adapt to the environment changes.

There are approaches using virtual machine (VM) migration including Cloudlet [4] and CloneCloud [5]. Application developers do not need to modify applications, but both approaches require transferring the entire or partial VM image from the smartphone to the cloud as well as complex profiling of the remote execution environment (*e.g.,* network/cloud performance) at run-time.

4.2 Process/Component Malleability

Maghraoui et al. [23] explores process-level malleability for MPI applications, which supports split and merge operations of MPI processes. Desell et al. [18] investigates component-level malleability using the SALSA programming language. They both support dynamic granularity change by splitting and merging processes and application components respectively; however, they require users to implement how to split and merge, whereas VM Malleability presented in this paper does not need cooperation from application programmers when the number of actor is larger than the number of VMs, but when the number of actors is smaller than the number of VMs an

actor has to split or make a clone to properly balance the load. COS supports actor split and merge functionalities, which we will explore further in the future.

4.3 VM Migration

VM migration has been investigated as a technique to dynamically change the mapping between physical resources and virtual machines. Clark et al. [15] proposes the pre-copy approach, which repeatedly copies the memory of a VM from the source destination host before releasing the VM at the source. Hines et al. [24] introduces the post-copy approach, which transfer the memory of a VM after its processor state is sent to the target host. Unlike regular Xen-based migration, Bradford et al. [25] realizes VM migration across wide-area network. While VM migration provides a lot of flexibility in managing virtual machines, the effectives of VM migration-based solutions are limited by the granularity of the VM instances.

4.4 VM Performance Analysis

Wang et al. [19] shows how granularity of virtual machines affects the performance in cloud computing environments especially for computation-intensive workloads. Based on their observations, they suggest that VM Malleability strategies can be used to improve the performance for tightly-coupled computational workload. Wang et al. [26] thoroughly investigates the impact of VM configuration strategies such as virtual CPUs and the memory size. They conclude tightly-coupled computational workloads are sensitive to the total number of VMs, and vCPUs per VM, and the memory size. It is a useful reference to VM management strategies and we will use their results in our research to fine-tune the performance of VMs.

5. Discussion

5.1 Conclusion

In this thesis, we have presented solutions to two categories of computation intensive applications: *i.e.,* (1) Moderately computation-intensive applications, which takes more than 10 seconds to process on a single smartphone, (2) Extremely computation-intensive applications, which takes a few minutes to a few hours to process on a single smartphone.

For the first category of application, we have proposed a model-based task offloading method using runtime profiling, which predicts the total processing time based on a simple linear model and offloads a task to a single server depending on the current performance of the network and the server. Since this model's simplicity greatly reduces the profiling cost at run-time, it enables users to start using an application without pre-computing a performance profile. Using our method, the performance of face detection for an image of 1.2Mbytes improved from 19 seconds to 4 seconds.

For the second category, we have presented a middleware framework, the Cloud Operating System (COS), as a back-end technology for smartphones. COS implements the notion of virtual machine (VM) malleability to enable cloud computing application to effectively scale up and down. Through VM malleability, virtual machines can change their granularity by using *split* and *merge* operations. We accomplish VM malleability efficiently by using application-level migration as a reconfiguration strategy. While VM migration does not require any knowledge of the guest Operating System or applications, our current VM malleability implementation requires migratable applications; however, this approach is still more general than common programming patterns such as MapReduce. Our experiments with a tightly-coupled computation show that a completely application-agnostic automated load balancer performs almost the same as human-driven VM-level migration; however, human-driven application-level migration outperforms (by 14% in our experiments) human-driven VM-level migration. This result is promising for future fully automated cloud computing resource management systems that efficiently enable truly elastic and scalable workloads.

We hope COS to be operated in a large data center someday, accept SALSA actors from a lot of smartphones to solve complicated tasks, and give a good experience to users.

5.2 Future Work

***For Light-weight Adaptive Task Offloading*:** In the future, we plan to apply our method to other applications to confirm the generality of the method. Candidate applications include but not limited to real-time face detection for video and real-time rendering of CFD (Computational Flow Dynamics). The face detection for video is a natural enhancement of the example we have done this time and is also expected to be pretty computationally intensive since we need to detect faces at very high frequency. Face recognition (comparing detected faces to a DB of images) is even more intensive in terms of computation and data requirements, and therefore it requires our task offloading approach to be practical. CFD is known as one of the most computationally intensive applications in general. Running CFD on smartphones sounds too complex, but it might be useful for games in terms of creating realistic motion of fluids and interaction with them. Another direction of enhancement could be on the server side. If an nVidia's or an AMD's graphics card is available on a server computer, we could utilize the GPUs by using CUDA or OpenCL and speed the process up greatly. We could also utilize tablets, such as Apple's iPad and Android Tablets, as servers since nowadays they use fast multicore processors. Offloading can also occur from a tablet to a server or a cloud. Cloud computing is suitable for task offloading purpose as well because it can provide great scalability in performance and give virtually infinite computing resources. Latency between a smartphone and clouds would vary depending on the location of the smartphone; therefore, light-weight adaptation on the smartphone shown in this paper would be critical. Using an actor-oriented programming language such as SALSA [2] is one way of implementing dynamically reconfigurable smartphone applications. In SALSA programming, an actor is the unit of execution. Consequently, an application written in SALSA is natively partitioned as a collection of actors and is suitable for dynamic partioning of the application at run-time. One research direction is to enhance

the model presented in this paper to support dynamic partioning using SALSA while keeping the model light-weight.

For Middleware for Autonomous Virtual Machine Malleability: We plan to keep working to improve the performance of autonomous actor migration. Also, in the future, we would like to support hybrid cloud model in COS to utilize the computing resource of public clouds such as Amazon EC2 and also develop a model to accept policies such as time-constrained, energy-constraint, and budget-constraint policy. It is also beneficial to support more fine-grained resource management in COS by configuring the number of vCPUs and the memory size of VMs. That will enable COS to control the performance and cost consumption more efficiently.

LITERATURE CITED

[1] K. Don (2011, September 1). *40 Percent of U.S. Mobile Users Own Smartphones; 40 Percent are Android* [Online]. Available: http://blog.nielsen.com/nielsenwire/online_mobile/40-percent-of-u-s-mobile-users-own-smartphones-40-percent-are-android/. (Date Last Accessed on October 31, 2011).

[2] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA", *OOPSLA '01: SIGPLAN Notices. ACM Object Oriented Programming Languages, Syst. and Applicat. Intriguing Technology Track Proc.*, vol. 36, no. 12, pp.20–34, Dec. 2001.

[3] C. Wang and Z. Li, "A computation offloading scheme on handheld devices", *J. Parallel and Distributed Computing*, vol.64, no.6, pp.740-746, June 2004.

[4] M. Satyanarayanan *et al.*, "The case for VM-based cloudlets in mobile computing", *IEEE Pervasive Computing*, vol. 8, no.4, pp.14-23, 2009.

[5] B. Chun *et al.*, "CloneCloud: elastic execution between mobile device and cloud", in *Proc. 6th European Conf. Comput. Syst.*, Salzburg, Austria, 2011, pp.301-314.

[6] S. Imai and C. A. Varela, "Light-weight adaptive task offloading from smartphones to nearby computational resources", in *Proc. 2011 Research in Appl. Computation Symp.,* Miami, FL, 2011, pp.146-151.

[7] Willow Garage. (2011, August 24). *OpenCV* [Online]. Available: http://opencv.willowgarage.com/wiki/. (Date Last Accessed on 10/31/2011).

[8] Y. Niwa (2011, April 17). *Using OpenCV on iPhone* [Online]. Available: http://niw.at/articles/2009/03/14/using-opencv-on-iphone/en/. (Date Last Accessed on 10/31/2011).

[9] M. Armbrust *et al.*, "Above the clouds: A Berkeley view of cloud computing," EECS Department, University of California, Berkeley, CA, Tech. Rep. UCB/EECS-2009-28, Feb. 2009.

[10] P. Barham *et al.*, "Xen and the art of virtualization", in *Proc. 20th ACM Symp. Operating Syst. Principles*, Lake George, NY, 2003, pp.164-177.

[11] E. L. Haletky, *VMware ESX Server in the Enterprise: Planning and Securing Virtualization Servers.* Upper Saddle River, NJ: Prentice Hall, 2008.

[12] RightScale. *Cloud Computing Management Platform by RightScale* [Online]. Available: http://www.rightscale.com/. (Date Last Accessed on 10/31/2011).

[13] Amazon Web Services. *Auto Scaling – Amazon Web Services* [Online]. Available: http://aws.amazon.com/autoscaling/. (Date Last Accessed on 10/31/ 2011).

[14] C. P. Sapuntzakis *et al.,* "Optimizing the migration of virtual computers", in *Proc. 5th USENIX Symp. Operating Syst. Design and Implementation*, Boston, MA, 2002, pp.377-390.

[15] C. Clark *et al.*, "Live Migration of Virtual Machines", in *Proc. 2nd USENIX Networked Syst. Design and Implementation*, Boston, MA, 2005, pp.73-286.

[16] J. G. Hansen and E. Jul, "Self-migration of operating systems", in *Proc. 11th ACM SIGOPS European Workshop*, Leuben, Belgium, 2004, pp.126-130.

[17] T. Wood *et al.*, "Sandpiper: black-box and gray-box resource management for virtual machines", *Comput. Networks*, vol. 53, no. 17, pp. 2923–2938, Dec. 2009.

[18] T. J. Desell *et al.,* "Malleable applications for scalable high performance computing", *Cluster Computing*, vol. 10, no.3, pp.323-337, 2007.

[19] P. Wang *et al.,* "Impact of virtual machine granularity on cloud computing workloads performance", in *Proc. 11th IEEE/ACM Int. Conf. Grid Computing*, Brussels, Belgium, 2010, pp.393-400.

[20] G. Agha, Actors: *A Model of Concurrent Computation in Distributed System.* Cambridge, MA: MIT Press, 1986.

[21] K. E. Maghraoui *et al.,* "The internet operating system: middleware for adaptive distributed computing," *Int. J. High Performance Computing Applicat., Special Issue Scheduling Techniques for Large-Scale Distributed Platforms*, vol. 20, no. 4, pp.467-480, 2006.

[22] E. Cuervo *et al.*, "MAUI: making smartphones last longer with code offload", in *Proc. 8th Int. Conf. Mobile Syst., Applicat., and Services,* San Francisco, CA, 2010, pp.49-62. R. K. Balan *et al.,* "The case for cyber foraging", in *Proc. 10th ACM SIGOPS European Workshop,* Saint-Emilion, France, 2002, pp.87-92.

[23] Y. Su, and J. Flinn, "Slingshot: deploying stateful services in wireless hotspots", in *Proc. 3rd Int. Conf. Mobile Syst., Applicat., and Services,* Seattle, WA, 2005, pp.79-92.K. E. Maghraoui *et al.*, "Malleable iterative MPI applications",

Concurrency and Computation: Practice and Experience, vol. 21, no. 3, pp.393-413, Mar. 2009.

[24] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning", in *Proc. 2009 ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Washington, DC, 2009, pp.51-60.

[25] R. Bradford *et al.*, "Live wide-area migration of virtual machines including local persistent state", in *Proc. 3rd Int. Conf. Virtual Execution Environments*, San Diego, CA, 2007, pp.169-179.

[26] Q. Wang and C. A. Varela, “Impact of cloud computing virtualization strategies on workloads’ performance”, in *Proc. 4th IEEE/ACM Int. Conf. Utility and Cloud Computing*, Melbourne, Australia, 2011, pp.130-137.