# Elastic Scalable Cloud Computing Using Application-Level Migration

Shigeru Imai, Thomas Chestna, Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
110 Eighth Street
Troy, NY 12180, USA
Email: {imais,chestt2,cvarela}@cs.rpi.edu

***Abstract*—We present the *Cloud Operating System (COS)*, a middleware framework to support autonomous workload elasticity and scalability based on application-level migration as a reconfiguration strategy. While other scalable frameworks (*e.g.*, MapReduce or Google App Engine) force application developers to write programs following specific APIs, COS provides scalability in a general-purpose programming framework based on an actor-oriented programming language. When all executing VMs are highly utilized, COS scales a workload up by migrating mobile actors over to newly dynamically created VMs. When VM utilization drops, COS scales the workload down by consolidating actors and terminating idle VMs. Application-level migration is advantageous compared to VM migration especially in hybrid clouds in which migration costs over the Internet are critical to scale out the workloads. We demonstrate the general purpose programming approach using a tightly-coupled computation. We compare the performance of autonomous (*i.e.*, COS-driven) versus ideal reconfiguration, as well as the impact of granularity of reconfiguration, *i.e.*, VM migration versus application-level migration. Our results show promise for future fully automated cloud computing resource management systems that efficiently enable truly elastic and scalable general-purpose workloads.**

## I. Introduction

Cloud computing is a promising computing paradigm that delivers computing as a service based on a pay-per-use cost model [1]. 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 partly enabled by virtualization technology such as Xen [2] or VMware [3]. Using virtualization, service providers can run multiple virtual machines (VMs) on a single physical machine, providing isolation to multiple user workloads, better utilization of physical hardware resources by consolidation of workloads, and subsequent energy savings [4], [5]. For those who have access to a private cloud environment, a hybrid cloud, which connects the private cloud to the public cloud, is an even better option in terms of scalability. When computational demands exceed the capacity of the private cloud, the hybrid cloud can let users scale out the computation by paying for extra resources on the public cloud.

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 [6] and Amazon's Auto Scale [7] 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 [8], [9], [10], [11]. Sandpiper [12] 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 VMs 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 [9], it incurs a significant load on the network.

On the other hand, it is possible to achieve load balancing and scalability with finer granularity for migration 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 and 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. Maghraoui et al. [13]

have developed a library to migrate processes autonomously for MPI applications; however, MPI applications need to be modified to support migration. Also, in the hybrid cloud environment, migrating a process from the private cloud to the public cloud is desirable. One solution to this problem is using SALSA (Simple Actor Language System and Architecture) [14]. SALSA 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 over the Internet. Applications composed of SALSA actors can be easily reconfigured at run time by using actor migration regardless of the network type (*e.g.*, LAN or WAN).

In this paper, we present the Cloud Operating System (COS), a middleware framework that supports 1) opportunistic creation and removal of VMs and 2) autonomous actor migration on VMs to enable cloud computing applications to effectively scale up and down. COS works as a PaaS (Platform as Service) infrastructure so that application programmers can focus on their problem of interest and leave resource management to COS. 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.

To enable elastic and scalable cloud computing applications, we have created the following design policies towards the construction of the COS middleware:

***Migration by actor granularity***: Application-level granularity should be a key to good load balancing. Also, we can expect application-level migration to take significantly less time compared to VM migration due to the footprint difference. Actor migration as supported by SALSA is a reasonable choice since the migration can be done transparently for application developers. 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 [15]. 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 interest and leave resource management to the underlying middleware. Furthermore, developers should not be constrained to specific programming patterns or application domains.

***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. Furthermore, it should be possible to migrate complete applications or only parts of them (*e.g.*, some of the component actors).

***Autonomous resource management***: 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 should autonomously create more VMs using available resources when all existing VMs are highly loaded and remove VMs to save energy and costs when it detects under-utilized VMs.

The rest of the paper is organized as follows. Section II describes how workloads scale by migrating actors. After explaining key technologies in Section III, Section IV introduces our distributed computation middleware, the Cloud Operating System. Section V compares the performance of VM-level and application-level migration. Section VI shows the results of autonomous reconfiguration experiments with a tightly-coupled workload. We present related work in Section VII, and finally conclude the paper with future work in Section VIII.

## II. Workload Scalability

The granularity of VM makes significant impact on workload performance in cloud computing environments [16]. 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 at run-time, 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 web servers using port 80 migrating into one VM. We can easily see that realizing completely transparent VM malleability is a challenging task.

However, we view migrating actors to newly created VMs as equivalent to splitting a VM and migrating actors from multiple VMs to one VM as equivalent to merging VMs. This notion of VM split and merge can be applicable to a hybrid cloud model as shown in Figure 1. We can not only scale up and down the computation within a private cloud by migrating actors, but also scale out and in to/from a public cloud seamlessly from the application point of view.

This VM granularity adjustment 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 automatically adjusts the granularity of VMs by migrating actors over dynamically created VMs.

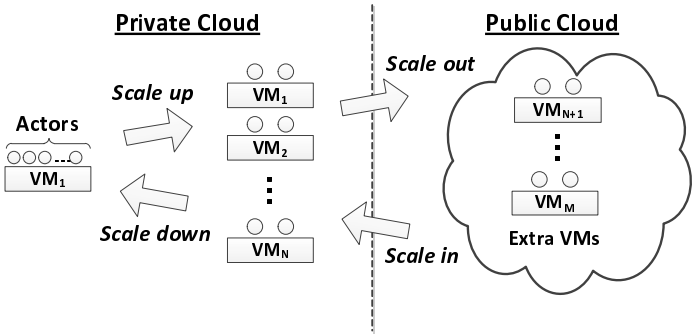


Fig. 1. Workload scalability over a hybrid cloud

## III. Technical Background

In this section, we introduce two key technological ingredients of our distributed computation middleware, SALSA [14] and the Internet Operating System (IOS) [17].

### A. 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 [18]. In addition to these features, it implements a universal naming model for actor migration and location-transparent message sending.

The name service, part of the SALSA runtime, binds a Universal Actor Name (UAN) to a Universal Actor Locator (UAL) and keeps track of its location, so that the actor can be uniquely identified by its UAN throughout its lifetime. This universal actor naming method enables high-level migration support and migration to any device hosting the SALSA runtime. SALSA greatly reduces the development cost of COS. Therefore, we use SALSA as the basis of our middleware and illustrate COS with applications written in SALSA.

### B. Internet Operating System (IOS)

IOS is a distributed middleware framework that provides support for dynamic reconfiguration of large-scale distributed applications through opportunistic load balancing capabilities. It has been shown that it is a scalable solution for distributed applications to well balance the load on cluster and grid environments [17], [19].

The load balancing algorithm used in IOS is a pull-based algorithm where an under-utilized IOS agent tries to 'steal' some work from other IOS agents. This process is executed in a decentralized manner, that is, one IOS agent passes a work-stealing request message to other IOS agents connected locally. Thus, this algorithm shows a good scaling performance, and it is stable: an overloaded node does not generate work-stealing requests.

If an IOS agent receives a work-stealing request message, it makes a decision whether to migrate an actor or a set of actors to a foreign node or not based on the following information.

Given:

- $R = \{r_0, ...r_n\}$: A set of resources
- $A = \{a_0, ...a_n\}$: A set of actors
- $\omega(r, A)$: A weight based on importance of the resource $r$ to the performance of a set of actors $A$, where $0 \leq \omega(r, A) \leq 1$ and $\sum_r \omega(r, A) = 1$
- $\alpha(r, f)$: The amount of resource available at foreign node $f$
- $\nu(r, l, A)$: The amount of resource used by actors $A$ at local node $l$
- $M(A, l, f)$: The estimated cost of migration of actors from $l$ to $f$
- $L(A)$: The average life expectancy of the actors $A$,

the predicted increase in overall performance $\Gamma$ gained by migrating $A$ from $l$ to $f$, where $\Gamma \leq 1$:

$$\Gamma = \sum_r (\omega(r, A) * \Delta(r, l, f, A)) - M(A, l, f)/(10 + logL(A)),$$

where

$$\Delta(r, l, f, A) = (\alpha(r, f) - \nu(r, l, A))/(\alpha(r, f) + \nu(r, l, A)).$$

When work is requested by $f$, IOS migrates actor(s) $A$ with greatest $\Gamma$, only if positive. If the migration fails for some reason (*e.g.*, network failure), the IOS agent simply waits for the next work-stealing message to come, which works fine since the sender of the work-stealing message keeps sending it as long as its utilization is low.

## IV. Cloud Computing Middleware

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

### A. System Architecture

COS uses the Xen hypervisor and Linux is used as a guest OS. It consists of several modules as shown in Figure 2. Each module is explained in order as follows.

- ***Actors***: Application actors are implemented in the SALSA programming language. They represent workloads to be executed 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.
- ***IOS Agent***: An IOS agent 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 described in Section III-B. As a result of migration, actors communicating often tend to be collocated in the same SALSA runtime [17].
- ***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 it detects consistently high or low CPU utilization.
- ***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 COS Manager.
- ***COS Manager***: The COS Manager analyzes reports from Node Managers and requests new VM creation or termination to 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.

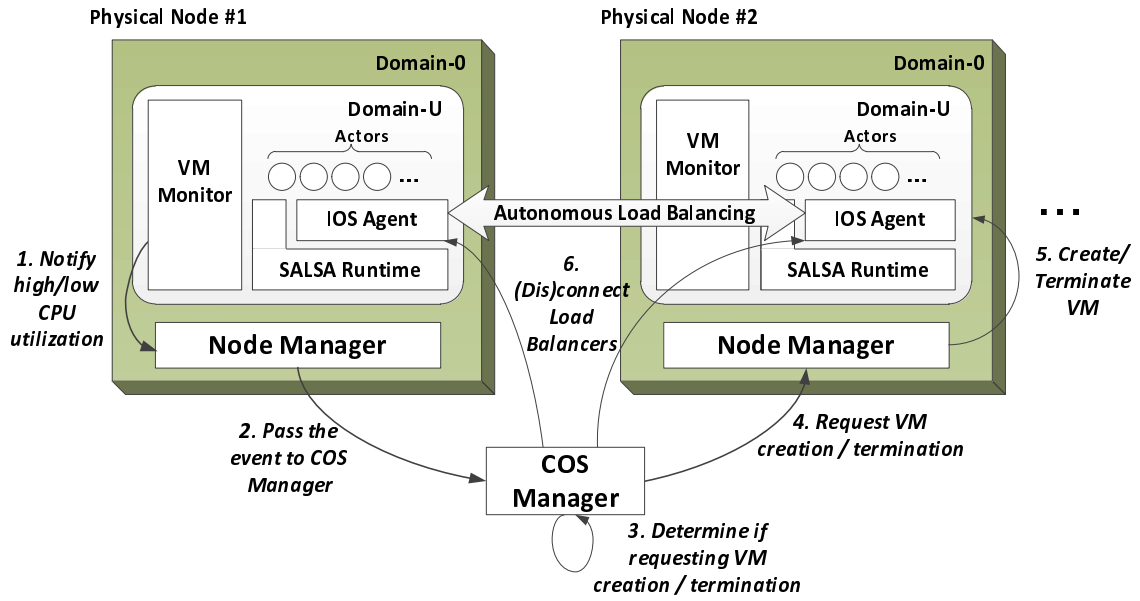


Fig. 2. System architecture of the Cloud Operating System

### B. Runtime Behavior of COS

Following, we show the steps of COS when scaling up a computation. Analogous 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 $k\_HIGH$ out of $n$ past utilization checks exceed the threshold just as Sandpiper [12] 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, it checks if it is feasible to create a new VM. Note that the COS Manager ignores frequent high CPU events to avoid too quick scaling of computing resources (Please see Section IV-C for detail).
- ***Step 4.*** If it is feasible to create a new VM, 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.
- ***Step 5.*** Upon receiving 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 an IOS agent 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 for the IOS agent.
- ***Step 6.*** When the COS Manager receives a VM creation event from a Node Manager, it connects available IOS agents using given IP addresses and port information, so that the IOS agents can start communicating with each other and balancing the load between them.

### C. Stable Creation and Termination of VMs

To avoid thrashing behavior due to frequent VM creation and termination, the following heuristics is implemented. We choose this threshold approach because it is simple yet effective to avoid thrashing behavior as we have experienced in [17].

- The COS manager decides to create a new VM only if it has received high CPU utilization events from all of the running nodes within a certain time range (*e.g.*, 10 seconds). The reason is that there may still remain under-utilized VMs and IOS agents may be able to balance the load and decrease the overall CPU utilization of VMs.
- Some workloads alternate between high and low CPU usage. 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 more difficult than creation.
- A VM Monitor sends a low CPU utilization event to a Node Manager only after it detects persistent low CPU utilization since it takes some time until an IOS agent gets some workload from other IOS agents.

## V. VM-level vs Application-level Migration

We conduct a preliminary experiment to confirm the characteristics of VM-level migration and application-level migration.

In this experiment, we use two machines. Each has quad core Opteron processors running at 2.27 GHz, connected via 1-Gbit Ethernet. We create VMs by using Xen-3.4.1 which works with Linux 2.6.18.8 and do regular migration between the two machines with the memory sizes from 0.5 Gbytes to 10 Gbytes. Note that no user applications are running at the time of experiments.

Figure 3 shows the result of VM migration time as a function of VM memory size.

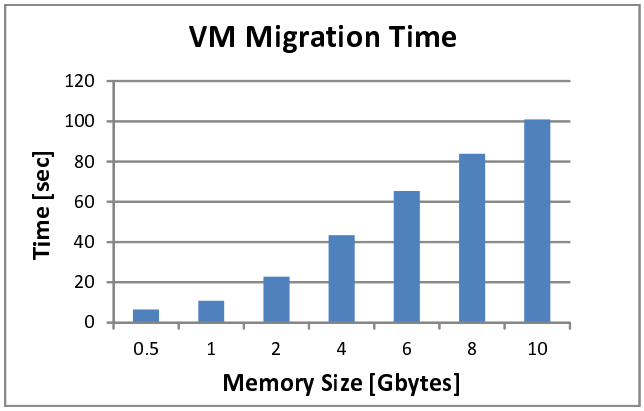


Fig. 3. VM migration time within a local network

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 150-160 ms and a 100 Kbytes actor takes 240-250 ms on a LAN, whereas the minimal actor takes 3-7 seconds and the 100 Kbytes actor takes 25-30 seconds on a WAN (migrating from Urbana, IL, USA to Paris, France to Tokyo, Japan). [14].

Furthermore, as illustrated in [20], the ratio between workload granularity and the number of processors has a significant impact on performance. As a consequence, VM-level coarse granularity has a disadvantage compared to the actor-level fine granularity in terms of load balancing. For example, in an experiment with two physical machines and an odd number of virtual machines (3 and 5), the execution time is 9.4% and 22.4% slower than using actor-based load balancing.

## VI. Autonomous Reconfiguration Performance

We have conducted experiments from two perspectives. In the first experiment, we test COS '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.

TABLE I
Test case scenarios for Experiment 1

| Test Case | vCPU/VM | PM | VM | Total vCPUs | Memory [MB/VM] |
|---|---|---|---|---|---|
| Fixed VM | 4 | 1 | 1 | 4 | 1024 |
| | 4 | 2 | 2 | 8 | 1024 |
| | 4 | 3 | 3 | 12 | 1024 |
| COS | 4 | dynamically determined at runtime | | | 1024 |

### A. Workload

The workload used is a Heat Diffusion problem, a communication-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. In both experiments, we run the problem for 300 iterations.

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.

We choose this problem as a benchmark because it is a general application that utilizes both network and CPU intensively, and also because it is known to be hard to scale.

### B. VM Configuration

Throughout the experiments, we use physical machines where each of them has quad core Opteron processors running at 2.27 GHz, and they are connected via 1-Gbit Ethernet. Also, every VM created during the experiments has four vCPUs and 1024 Mbytes of memory.

### C. Experiment 1 - Test COS with Single User

*1) Experimental Setup:* In this experiment, only one user runs the Heat Diffusion problem on COS. We also compare the execution time of the 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 and create only one VM per physical machine. 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. Table I summarizes the test case scenarios used in Experiment 1.

*2) Results:* ***VM CPU Utilization:*** As shown in Figure 4, 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. Since the actors are gradually migrated by IOS agents from VM1 to VM3, not from VM2, the CPU utilization of

VM3 increases whereas the CPU utilization of VM1 decreases almost by half. Therefore, COS successfully creates VMs when extra computing power is demanded and in steady-state conditions it balances the load among the different VMs.

***Throughput:*** Figure 5 depicts the throughput of the Heat Diffusion simulation. It is clear that the throughput gets higher as the number of VMs and accumulated VM CPU utilization (as shown in Figure 4) increases.

***Execution time:*** Figure 6 shows the relationship between the number of vCPUs and the execution time. COS uses 9.002 vCPUs on average, whereas 1, 2, and 3VMs cases 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 of 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.

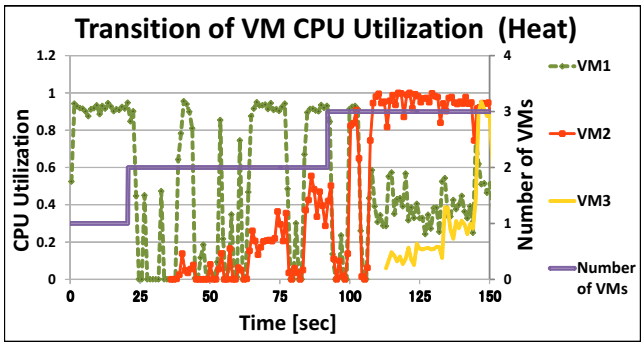


Fig. 4. VM CPU utilization from Heat Diffusion problem running on the Cloud Operating System

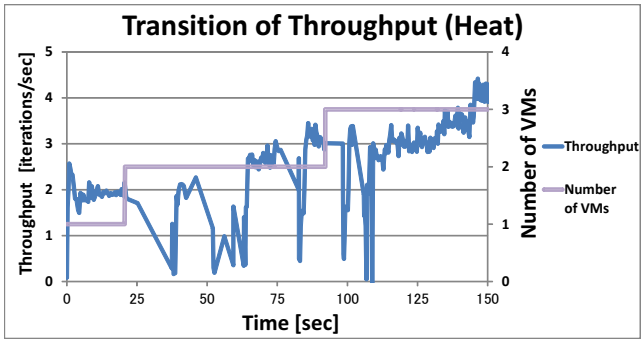


Fig. 5. Throughput of the Heat Diffusion problem running on the Cloud Operating System

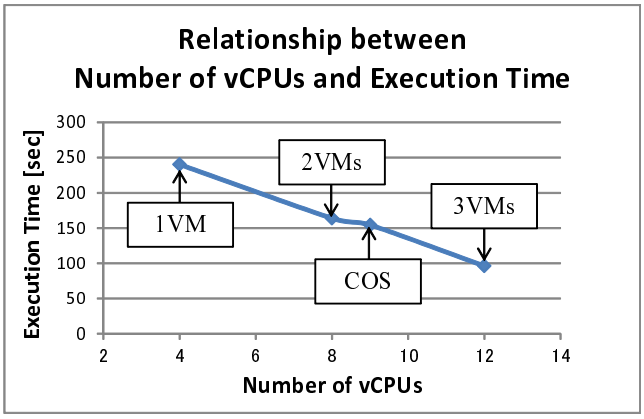


Fig. 6. Throughput of the Heat Diffusion problem running on the Cloud Operating System

### D. Experiment 2 - Compare COS with Ideal Migrations

*1) Experimental Setup:* In this experiment, we compare the performance of autonomous actor migration used in COS with ideal migration methods. We consider two users submitting their 8 actors each for the Heat Diffusion problem over two physical machines (PM1 and PM2). Initially, User 1 submits his 8 actors on PM1, and then 60 seconds later, User 2 submits another 8 actors on PM2. How those actors are actually assigned on VMs depends on test case scenarios described in Table II. We test autonomous actor migration along with two ideal cases where migrations are initiated by a human as described in the next section.

*2) Ideal Migration:* In the cases of ideal human-driven VM migration and actor migration, a human monitors the resource usage on both physical machines and initiates migration to get the workload balanced as examples shown in Figure 7. Example 1 shows a case where User 1 has two VMs, where each VM has 4 actors, on PM1 and then he migrates the second VM ($\mathrm{VM2_{U1}}$) to PM2. Example 2 illustrates a situation where User 2 newly submits 8 actors on his two VMs on PM1. In this case, the human migrates the first VM of User 1 ($\mathrm{VM1_{U1}}$) to PM2 to collocate the VMs of User 1, which is beneficial for tightly-coupled workloads. Same policy applies to ideal actor migrations.

*3) Results:* The execution time results of each test scenario are shown in Figure 8. We measure the execution time required for all tasks to finish. Autonomous actor migration performs almost the same as ideal VM migration, whereas the ideal actor migration is the fastest of the three methods and is about 14% faster than the autonomous actor migration and the ideal VM migration. This clearly shows the benefit of actor migration compared to VM migration approach. IOS agents used by COS do not know anything about applications such as communication patterns or dependencies between modules; therefore the IOS agents have 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 ideal actor migration

TABLE II
TEST CASE SCENARIOS FOR EXPERIMENT 2

| Test Case Scenarios | Description |
|---|---|
| VM migration (Ideal) | *A human decides when & where to migrate VMs to achieve optimal load balancing* |
| | • User 1 has two VMs ($VM1_{U1}$ and $VM2_{U1}$) on PM1, and User 2 has two VMs ($VM1_{U2}$ and $VM2_{U2}$) on PM1.<br>• Initially, User 1 submits 4 actors on $VM1_{U1}$ on PM1 and another 4 actors on $VM2_{U1}$ on PM1. Right after the VM submissions, $VM2_{U1}$ is migrated to PM2 for the best use of resources (See Figure 7a).<br>• 60 seconds later, User 2 submits 4 actors on $VM1_{U2}$ on PM1 and another 4 actors on $VM2_{U2}$ on PM1. Right after the VM submissions, $VM1_{U1}$ is migrated to PM2 to be collocated with $VM2_{U1}$ (See Figure 7b). |
| Actor migration (Ideal) | *A human decides when & where to migrate actors to achieve optimal load balancing* |
| | • There is one VM (VM1) on PM1 and another one (VM2) on PM2. Users do not have dedicated VMs, but share VM1 and VM2.<br>• Initially, User 1 submits 8 actors on VM1 on PM1. Right after the actor submissions, 4 out of 8 of his actors are migrated to VM2 on PM2 for the best use of resources.<br>• 60 seconds later, User 2 submits 8 actors on VM1 on PM1. Right after the actor submissions, User 1's remaining 4 actors are migrated to VM2 on PM2 to be collocated with his another 4 actors. |
| Actor migration (Autonomous) | *VM allocation and actor migration is automated by COS* |
| | • Initially, User 1 submits 8 actors on a VM allocated on PM1.<br>• 60 seconds later, User 2 submits another 8 actors on the same VM allocated on PM1. |

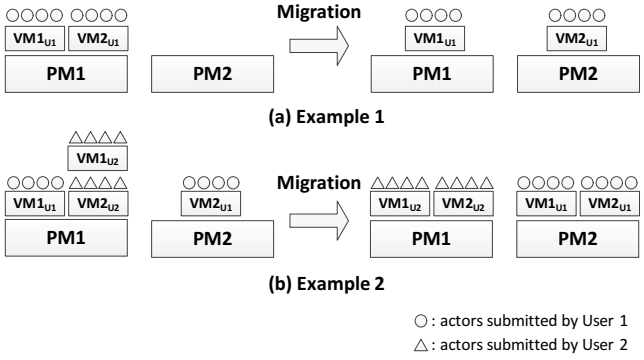


Fig. 7. Examples of ideal VM migration initiated by human

is able to migrate multiple actors to the right place at the right time. The reason of the ideal VM migration's moderate performance is due to the cost of VM migration. As we have shown in Section V, migration of a 1024 Mbytes memory VM takes about 10 seconds. During the migration time, communication between actors suffers significantly, therefore the performance is degraded. Clearly, human-driven resource management is not scalable, or even feasible in dynamic environments; so these results with autonomous actor migration are quite promising.

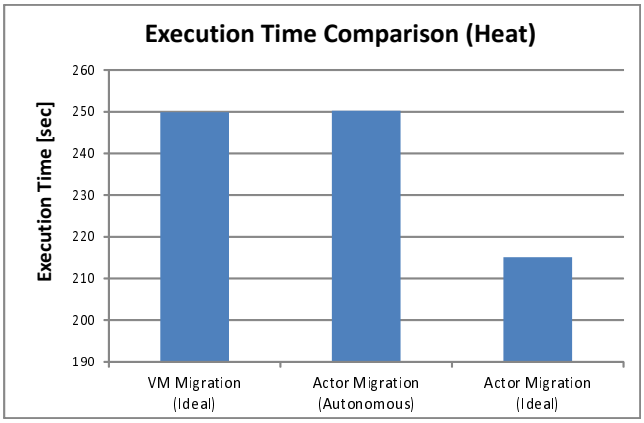


Fig. 8. Execution time comparison of Heat Diffusion problem

## VII. RELATED WORK

Kalé et al. [21] achieves MPI-process level malleability by using AMPI (an adaptive implementation of MPI), which can be reconfigured at run-time. Maghraoui et al. [13] explores process-level malleability for MPI applications, which supports split and merge operations of MPI processes. Desell et al. [20] investigates component-level malleability using the SALSA programming language. They both support dynamic granularity change by splitting and merging of processes and application components respectively; however, they require users to implement how to split and merge. The middleware presented in this paper does not need cooperation from application programmers when the number of actors 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. IOS supports actor split and merge functionalities, which we will explore further in the future.

Wang et al. [16] 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 dynamic VM granularity strategies can be used to improve the performance for tightly-coupled computational workload. Wang et al. [22] thoroughly investigates the performance 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, vCPUs per VM, and the memory size. It is a useful reference to VM management strategies and we plan to use the results in our future research to fine-tune the performance of VMs.

## VIII. CONCLUSION AND FUTURE WORK

This paper introduced a middleware framework, the Cloud Operating System, that supports autonomous cloud computing workload elasticity and scalability based on application-level migration. To scale a workload up (or out,) COS creates new VMs (in a public cloud) and migrates application actors to

these new VMs. To scale a workload down (or in,) COS consolidates actors into fewer VMs and terminates idle VMs.

While VM migration allows for dynamic cloud computing workload reconfigurations, and it does not require any knowledge of the guest O.S. or applications, its performance is prohibitive in hybrid clouds requiring potentially migrating several gigabytes over the wide area network. It is also an ineffective approach in applications using mobile devices (*i.e.*, smart phones) given their constrained resources. Higher-level approaches to scalability (*e.g.*, MapReduce [23] or Google App Engine [24]) impose specific programming patterns or APIs on applications making these approaches less general.

COS only requires cloud applications to contain migratable components and does not impose any further restrictions on workloads. Using application-level migration, workloads can be elastic and scalable autonomously through middleware-driven dynamic reconfigurations. Preliminary experiments with a tightly-coupled computation in a private cloud show that a completely application-agnostic automated load balancer performs almost the same as optimal (*i.e.*, human-driven) VM-level migration. Through ideally-triggered application-level migration, we could get even better performance (14% in our experiments).

We plan to keep working to improve the performance of COS-driven workload reconfigurations. We also plan to experiment with more heterogeneous and realistic workloads. We will develop a model to accept high-level policies such as time-constrained, budget-constrained, and energy-constrained as well as QoS parameters (such as throughput or deadlines) [25] to drive reconfigurations in hybrid clouds. Future work also includes exploring automating more fine-grained resource management in COS by dynamically reconfiguring the number of vCPUs and the memory size of VMs (see [22] for the potential performance impact of these reconfigurations performed manually.) Finally, future work includes connecting mobile device applications to hybrid clouds (*e.g.*, see [26] ) using application-level migration to gain elasticity, scalability, and efficiency, in a general-purpose programming framework.

## ACKNOWLEDGMENT

We would like to thank Naveen Sundar Govindarajulu for his valuable feedback. This research is partially supported by Air Force Office of Scientific Research Grant No. FA9550-11-1-0332.

## REFERENCES

[1] 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.*, vol. UCB/EECS-2009-28, Feb 2009.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization." in *SOSP'03*, 2003, pp. 164–177.

[3] E. L. Haletky, *VMware ESX Server in the Enterprise: Planning and Securing Virtualization Servers*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall, 2008.

[4] K. H. Kim, A. Beloglazov, and R. Buyya, "Power-aware provisioning of cloud resources for real-time services," in *Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science*, ser. MGC '09. New York, NY, USA: ACM, 2009, pp. 1:1–1:6.

[5] C.-C. Lin, P. Liu, and J.-J. Wu, "Energy-efficient virtual machine provision algorithms for cloud systems," in *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, ser. UCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 81–88.

[6] Rightscale, "Cloud Computing Management Platform by RightScale," http://www.rightscale.com/.

[7] Amazon Web Services, "Auto Scaling - Amazon Web Services," http://aws.amazon.com/autoscaling/.

[8] 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.*, vol. 36, pp. 377–390, December 2002.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines." in *NSDI'05*, 2005, pp. 273–286.

[10] Hansen, J. Gorm, and E. Jul, "Self-migration of operating systems," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, ser. EW 11. New York, NY, USA: ACM, 2004.

[11] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 25–25.

[12] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines." *Computer Networks*, pp. 2923–2938, 2009.

[13] K. E. Maghraoui, T. Desell, B. K. Szymanski, and C. A. Varela, "Malleable iterative MPI applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 3, pp. 393–413, March 2009.

[14] C. A. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, vol. 36, no. 12, pp. 20–34, Dec. 2001.

[15] C. Varela, "Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination," Ph.D. dissertation, U. of Illinois at Urbana-Champaign, 2001.

[16] 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)*, Brussels, Belgium, October 2010, pp. 393–400.

[17] 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*, vol. 20, no. 4, pp. 467–480, 2006.

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

[19] K. E. Maghraoui, "A framework for the dynamic reconfiguration of scientific applications in grid environments," Ph.D. dissertation, Rensselaer Polytechnic Institute, 2007.

[20] T. Desell, K. E. Maghraoui, and C. A. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, pp. 323–337, June 2007.

[21] L. V. Kalé, S. Kumar, and J. Desouza, "A malleable-job system for timeshared parallel machines," in *Proceedings of the 2nd International Symposium on Cluster Computing and the Grid (CCGrid 2002*, 2002, pp. 230–237.

[22] Q. Wang and C. A. Varela, "Impact of cloud computing virtualization strategies on workloads' performance," in *4th IEEE/ACM International Conference on Utility and Cloud Computing(UCC 2011)*, Melbourne, Australia, December 2011.

[23] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters." in *OSDI'04*, 2004, pp. 137–150.

[24] Google, "Google App Engine," https://developers.google.com/appengine/.

[25] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds," *Future Generation Comp. Syst.*, pp. 861–870, 2012.

[26] S. Imai and C. A. Varela, "Light-weight adaptive task offloading from smartphones to nearby computational resources," in *Research in Applied Computation Symposium (RACS 2011)*, Miami, Florida, November 2011.