Optimizing Client-side Resource Utilization in Public Clouds

Swapnil Haria, Mihir Patil, Haseeb Tariq, Anup Rathi, Michael Swift
Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI, USA
{swapnilh, mihir, haseeb, amrathi, swift}@cs.wisc.edu

Abstract—**Public clouds such as Amazon's Elastic Cloud Compute (EC2) and the Google Compute Engine are being increasingly used by organizations as well as individuals to rapidly set up infrastructure to deploy their applications. While these cloud providers promise flexibility and performance to users, efficient and cost-effective management of Virtual Machine (VM) clusters is difficult to achieve in practice. The coarse granularity of resource allocation and dynamically varying resource requirements of applications result in excess costs due to VM under-utilization or performance degradation on undersized VMs. The growing number of applications being moved to such cloud environments magnifies the scope of this problem.**

We propose a Client-side Resource Manager which uses on-demand application migration and dynamic migration policies to improve VM utilization, and extract the best possible performance from a VM cluster for a given operating cost. Two migration policies are developed and evaluated using a new simulator, created by us to rapidly validate migration policies over longer time-frames. We demonstrate improved cost-efficiency of 25% over the conventional approach.

I. Introduction

Cloud computing is a promising new technology aimed at helping clients decouple software services from hardware infrastructure. In April 2015, Amazon reported that its Amazon Web Services (AWS) division had hit annual revenues of $5 billion, and is growing rapidly at about 49% each year [4]. This spectacular rise of cloud platforms can be attributed to on-demand availability of hardware, support for various usage patterns and elimination of infrastructure management costs [6].

In the Infrastructure as a Service (IaaS) model dominant today, these end-users lease resources in fixed-sized chunks of compute, memory, and I/O units (such as instances in the Amazon EC2 cloud). However, reasoning about the resource requirements of applications is tough, especially for front-end applications like web servers whose requirements vary widely with the incoming load. Real-world observations demonstrate up to 5x CPU and 2x memory resource over-provisioning for applications in analogous situations like submitting jobs to a datacenter [11]. This benefits cloud operators, who use techniques like memory sharing and overcommitment as well as virtual CPU multiplexing to improve the utilization of their physical infrastructure [22].

The Resource as a Service model [2] has been proposed to increase the granularity of resource allocation in the cloud environment, primarily to ensure greater value for the client's money. While this helps avoid the cost of paying for unused resources, it is hard for developers and users to reason about increasingly finer amounts of resources. Other solutions include hybrid resource provisioning strategies [8, 10] to optimize the cost efficiency of cloud computing for the end user. These are limited solutions as these tackle the problem of choosing between on-demand and reserved resources for incoming jobs to minimize running costs. However, VM utilization needs to be considered to guarantee cost efficiency, especially for long-running VM clusters in the cloud.

We believe that the problem is not in the IaaS model itself, but in the lack of control available to the end users to optimize their purchased resources. Our approach tackles this through the use of on-demand application level migration. Application-level migration or process migration as it used

to be called, has proven to be tough to implement practically due to issues of residual dependencies and transparency [19]. Fortunately, cloud-based operating systems like Drawbridge [20] and OSv [14] have increased process isolation and narrower process-kernel interactions, which results in better application mobility. These OSes had been initially proposed to optimize the footprint of operating systems in the cloud, by building on the library OS model [5, 12]. As a result, our implementation is aimed for usage in such environments, and leverages the improved application mobility to facilitate efficient application-level migration.

In this paper, we propose CSRM, a client-side resource manager for optimizing VM utilization and improving the cost-efficiency of cloud computing for the end-user. CSRM uses on-demand application-level migration to shift processes to larger virtual machines (VMs) during peak demands (scale-up), or relocate processes to smaller VMs during times of light load (scale-down). This avoids the need to kill and restart processes on larger VMs on exceeding the estimated memory demand, for which existing progress gets discarded. The resource manager is also responsible for managing the number and size of VMs (scaling out or in) needed to adequately satisfy the performance requirements of these jobs.

Our contributions can be summarized as follows-

- Discussion of the opportunities and challenges in streamlining public cloud usage,
- Conception of the CSRM framework to improve VM utilization and cost-efficiency,
- Development of a simulator to rapidly evaluate the various migration policies,
- Description and evaluation of two migration policies.

This paper is structured as follows. Section II describes the feasibility issues associated with process migration, and explains how cloud OS environments are conducive for such migration. In Section III, we discuss the challenges in efficiently managing a VM cluster in a public cloud. The software architecture of CSRM is presented in Section IV, and the migration policies are detailed in Section V. Section VI describes our methodology, and the various migration policies are evaluated in Section VII. Section VIII is a summary of the related work, and we outline future directions while concluding in Section IX.

II. Background

A. Process Migration

Process migration is the act of transferring a process along with its all data and state to another machine and resuming execution on that machine. There are many applications of process migration such as load balancing in clusters, providing fault resilience and exploiting data locality in NUMA and other distributed systems [19]. Unfortunately, there are many challenges in adding support for process migration.

The major issues in supporting application migration are the complexity of the implementation and handling the dependencies of an application with the operating system. The level at which the support is added, kernel level or user space, has consequences for the complexity, performance and transparency of the migration. User space implementations are simpler, and have better knowledge of the application's behavior but can suffer from reduced transparency and performance. Transparency requires that neither the migrated tasks nor any other interacting tasks notice the effects of the migration.

Communications can be delayed but not disrupted, and IO channels and file descriptors should be preserved across migrations. Transparency introduces complexity, and there are trade-offs between transparency and performance as well. Particularly for networked applications, redirecting communication through old links after migration leads to residual dependencies. The forwarding costs can get significant with increasing number of migrations. Furthermore, different applications are impacted differently by migration. Migration delays are tolerable for long running applications but could be prohibitive for short running and latency critical applications. As discussed in the next section, some of these issues are mitigated in library OSes.

B. Cloud Operating Systems

Currently, operating systems in cloud computing environments can be categorized in two groups. The majority of the virtual machines in the cloud run existing general purpose operating systems (either unmodified or slightly modified in the case

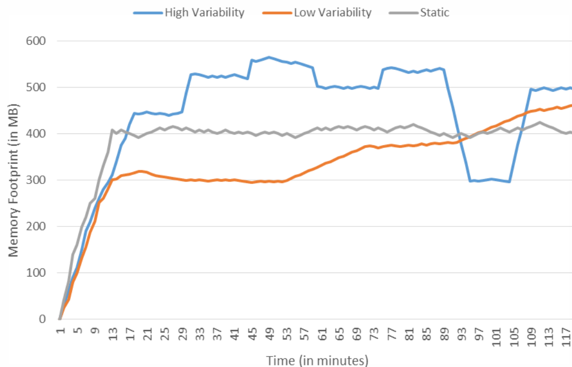

Fig. 1: Representative consumption patterns for different workload types

of paravirtualization) like Windows or Linux. The other end of the spectrum involves minimalist operating systems like Drawbridge [20], OSv [14] and MirageOS [18], which are tailored for the cloud setup. OSv and MirageOS are single address-space kernels streamlined for executing a single process. Drawbridge, a prototype of a Windows-based library OS enables fine-grained packing of self contained applications.

These were initially proposed as an alternative to VMs, by offering isolation and mobile execution environments with a smaller resource footprint. However, our belief is that Drawbridge is better suited as a guest OS running on top of VMs to fit into the current IaaS model. This approach bears resemblance to the nested virtualization model proposed by the turtles project [7]. Drawbridge relies on maintaining minimal application state in the kernel and allows a narrow set of interactions between the application and the kernel. As a result, Drawbridge enables the migration of applications across machines. Our proposal relies on support for efficient process migration, and hence we target our implementation for VMs running Drawbridge-like OSes.

III. Problem

A. *Challenges in streamlining cloud usage*

Currently, there are three ways for clients to provision resources for their applications in the cloud: reserved, on-demand and spot instances [3]. Reserved instances are the most cost-effective in terms of price per resource but require long-term commitments. Greater flexibility is provided by on-demand instances which do not require any such commitment, and have a fixed albeit steeper hourly rate. Spot instances have the potential for the most savings as the user can bid any price for cloud resources, but these instances get cloud resources only during times of low demand. Thus, the developer needs to figure out the right balance between cost and duration of reservation for his workloads to obtain maximum savings. This work focuses on the management of on-demand instances, although ideas for incorporating reserved instances into our policies are briefly touched upon in Section IX.

The other major decision faced by the user of a cloud service is the configuration of the virtual machine on which to deploy an application. The options range from general-purpose VMs with 1 core and 1GB RAM to server-class machines with 36 cores with 244 GBs of RAM [1]. This choice is motivated by the compute and memory requirements of the application under consideration as well as financial considerations based on the budget and the cloud provider's pricing model. However, it is particularly difficult to determine the resources needed by an application given the complexity of modern code bases. In addition, the load of user-facing services like web servers varies widely within a day, while the load of analytics tasks depends on their complexity and dataset size.

Previous literature has reported real-world data collected from users estimating resource requirements for jobs being submitted to a production cluster at Twitter [11]. This study showed that 70% of users overestimated requirements by 10x while 20% of users underestimated requirements by 5x. Given that users submitting jobs to datacenters or deploying applications on VMs in the cloud need to make similar decisions, these numbers can be taken as indicative for cloud scenarios as well. In public clouds, undersized reservations lead to poor application performance, while over-sized reservations lead to low resource utilization and poor cost-efficiency. These issues get intensified when the user has multiple applications running concurrently in multiple VMs.

[1]Based on Amazon EC2 instance types- http://aws.amazon.com/ec2/instance-types/

B. Efficient VM cluster management

Managing VM clusters with multiple running applications presents several challenges. Effective load balancing requires transparent process migration to move applications across VMs. As described in Section II-B, upcoming cloud environments offer greater application mobility. As a result, we safely assume that process migration is available in our target environments.

With process migration in place, the next obstacle is determining when such migrations be initiated and which processes to migrate. At a high-level, migrations can be triggered for two reasons. Desired performance can be achieved in loaded systems by scaling up (moving the process to a bigger VM) or scaling out (improving the process to VM ratio by creating more VMs). Under lightly loaded situations, VM utilization and thus cost-efficiency can be improved through scaling down (moving the process to a smaller VM) or scaling in (deleting VMs and co-locating greater number of applications on each VM). It is essential to timely identify scenarios when opportunities arise for improving performance or costs by triggering any of the above migrations.

The next hurdle comes in the form of the choice of application selected for migration and the destination VM. Applications running in cloud environments can be classified in three distinct categories based on their load patterns- static, low variability and high variability workloads [11]. Representative resource requirement patterns for these three workload scenarios is shown in Figure 1. Static applications can be moved to suitably sized VMs to improve utilization, but this may not be feasible as VM configurations are determined by the cloud provider. Also, larger VM instances are generally priced at a lower cost per resource ratio.

Both low and high variability workloads have dynamically varying resource requirements. Thus, the VM instance type determined as the destination for the process may not be suitable for the process in the future, resulting in greater migrations. Even though process migration is assumed to be possible, it is an expensive operation in terms of the time taken by the migration itself. In our experimental setup, application migration takes about 10-30 seconds depending on the memory footprint of the application. As certain kinds of workloads (like servers) have strict upper bounds on total downtime, it is imperative that the same application is not migrated often. Hence, the number of migrations for each process need to be monitored and more importantly, considered by the migration policy. Finally, financial considerations play an important role in VM cluster management. According to the budget available, the hourly cost of the total cloud operations may need to be capped off at an upper limit, while also ensuring that the user obtains optimal performance at that price point. This is particularly important for front-end services like application and web servers which should scale up in performance to satisfactorily service incoming traffic, but only till it is financially feasible to sustain it. In the absence of this cap, web servers could be faced with hefty bills on being Slashdotted [1] or hit by denial-of-service (DoS) attacks, when the server is temporarily flooded with requests.

IV. Software Architecture

The underlying infrastructure in a public cloud setting consists of heterogeneous physical host machines on which users can spawn virtual machines as needed. The VMs that are available to the users also differ in their specifications like virtual CPUs and main memory. The user of the public cloud service only sees and interacts with the set of VMs that he has spawned. In this paper, we use the term “system” to describe the set of VMs belonging to the user. CSRM provides a framework for the users to maximize the output in terms of VM utilization and performance they can get while optimizing the total costs payable to the cloud service provider.

The design of CSRM as shown in Figure 2 follows a two-tiered model.The Migrator module maintains global state and directs the Observers. Each Observer carries out the orders of the Migrator and also reports current status back to the Migrator. There are two reasons for this choice. It is easier to maintain a global view of the system at a central controller. In a distributed setting, all Observers would have to keep the whole or portions of the global state, which is harder to achieve and increases storage overheads. Also, our approach reduces the amount of control messages needed to communicate state as each Observer

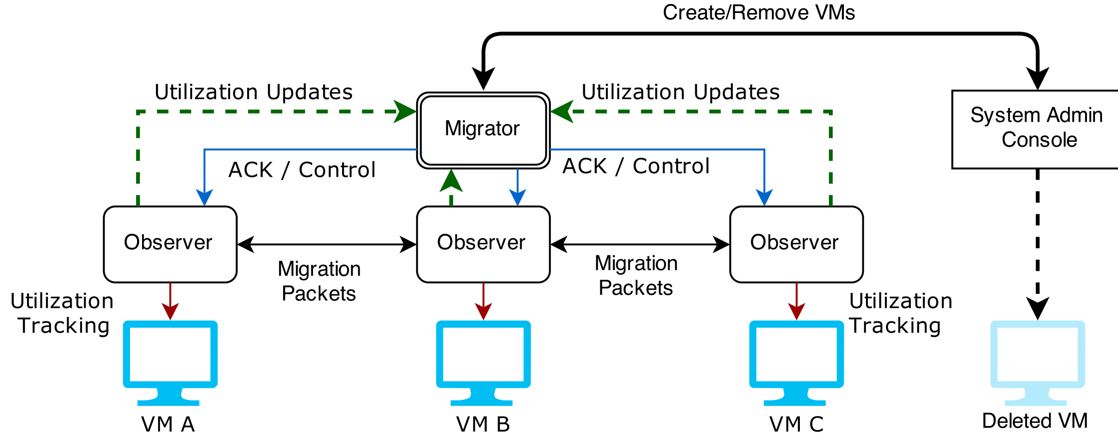

Fig. 2: Software architecture of the CSRM

only communicates with the Migrator regularly. Observers still need communicate with each other to carry out the actual process migration, which is an infrequent occurrence. The detailed functioning of the Observers and the Migrator are described below.

1) Observer: An Observer runs as a thread on every VM that the user creates. This thread starts when a VM is booted up and runs in the background as a daemon. It continuously monitors the CPU and memory utilization of the VM and sends periodic messages to the Migrator. Each message contains a VM identifier, a process identifier and resource requirements of all the processes running on that VM. While the Migrator is responsible for initiating process migration when required, Observers handle the actual migration. It checkpoints the process state and transfers this data to the destination VM. The Observer on the destination VM receives this data, saves the process context and resumes process execution.

2) Migrator: A Migrator thread runs in the background on one of the VMs in the system. The Migrator thread typically runs on the first VM started, although this is not strictly necessary. It collects information from the Observer thread running on each VM and collates it to obtain a global view of the system. Depending on the state of the system (heavily or lightly loaded) as well as the migration policy installed on the Migrator, it determines the need for any migrations. When a migration is indicated, the policy also identifies a suitable process for migration and the destination VM, and communicates this via control messages to the source VM. While the lone Migrator may emerge as a bottleneck, it is unlikely as Observers communicate rather infrequently with the Migrator, typically once every minute.

In addition, the Migrator periodically sends out heartbeat messages to all VM nodes. This helps Observers on newly created VMs identify the Migrator. If the policy advocates closing the VM on which the Migrator is located, the Migrator co-ordinates with an Observer thread on a randomly chosen VM to launch another Migrator thread there with the correct global state. A heartbeat message then helps all the Observers identify the new Migrator. Migrator also allows the user to set policy parameters like limits on the hourly operating cost or the maximum number of migrations per process per day. Scientific workloads are relatively insensitive to migrations, while servers have an availability criteria. As such, the user is best-placed to define the limit on the number of migrations, depending on the nature of the application. Lastly, it communicates with the System Administrator via a System Admin Console to create/delete VMs as required.

V. MIGRATION POLICIES

With the mechanisms for reporting usage statistics and capabilities to change the state of system in place, we look at the migration policies needed to drive the system towards an optimal state. For simplicity, we consider memory footprint as the only resource considered by any policy.

From the user's perspective there are two observable metrics to care about, namely the running costs of VM instances, and the quality of service (QoS) provided to each application. QoS is gauged by observing two key indicators that directly influence performance - the downtime incurred by an application due to migration and memory overcommitment on an application's VM. Memory overcommitment in a VM occurs due to the use of a smaller VM configuration than required for peak performance. Memory overcommitment also represents VM utilization upto a value of 100%, beyond which the VM utilization stays at 100% but each application observes a performance degradation. This would occur when the total operational costs per hour are be bounded by a user specified upper limit. Similarly, the maximum allowable downtime on a per-application basis can also be supplied by the user.

There exists a trade-off between QoS provided and the savings made by consolidating applications onto fewer and cheaper VM instances. Aggressive consolidation could lead to performance degradation and excessive migrations. So the central principle driving these policies is to keep total VM utilization between certain upper and lower thresholds. The absolute limits set by the user on migration downtime and running costs take precedence over other policy decisions. If the limits are too restrictive to allow the system to scale up or scale out, the overall system performance suffers. When a VM's utilization exceeds the upper threshold, processes are migrated to bigger or newly created VM instances to scale up/out the cluster. If a VM's utilization goes below the lower threshold, there is opportunity to scale down/in the system by moving to less powerful VMs or shutting down VMs to reduce the operational cost.

The application relocation policies can be divided into three main steps-

1) Identifying processes for migration.
2) Redistribution among the set of existing VMs, if possible.
3) Creating/Deleting VM instances to accommodate outstanding processes from step 2.

There are two different application layout schemes to consider when developing the policies. We describe one migration policy applicable to each of these schemes in this section. For both of these policies, the lower VM utilization threshold is 50% and the upper threshold is 95%. These were empirically chosen looking at results from our simulator, described in Section VI B.

A. *Single Application Per VM*

This is the trivial case in which each application runs in a separate VM. On start up, every application is allocated to a new VM instance with the best-fitting configuration. This best fit policy is then used to handle over and under-utilization events by upgrading or downsizing the VM as per the resource demand of the application. This will be referred to as One-Policy in the subsequent sections.

B. *Multiple Applications Per VM*

In this scenario, there can be multiple applications allocated on every VM instance. There are various ways of deciding which processes to migrate in over or under-utilized scenarios. Selecting processes in increasing order of memory demand and random selection are a couple of options. We can also tag applications according to their resource usage patterns as seen in Figure 1. With this classification, we can prioritize applications for migration based on their resource usage patterns. We choose the process with the highest memory demand to migrate to reduce the number of migrations and ease of implementation.

In the redistribution step the processes selected for migration are checked against the existing set of VMs to try and share resources without over-utilizing any single VM instance. Target VM selection is done on a per process basis beginning with the process with the highest memory requirement. If a target VM is found, the process is migrated to that VM and it joins the runnable processes on that VM. Any process for which a target VM is not found is marked as a leftover. These leftover processes would require new VMs to be created.

The resulting problem is to figure out the most economical configuration and number of new VMs to house these applications. This is similar to the server consolidation problem where multiple VM instances need to be packed on physical machines. The server consolidation problem has been shown to reduce to the multi-dimensional bin packing problem [21], which is known to be NP-hard. Consequently, we resort to the simpler first-fit policy considering the largest outstanding process to get a starting VM configuration. We then try to fit all processes on this configuration as well as the configuration immediately larger than the starting one, and iterate similarly till all outstanding processes are accommodated. The costs incurred for each of resulting configurations are compared, and the cheapest one is selected. This will be referred to as Multiple-Policy in the subsequent sections.

For scaling down the system, the goal is to shut down under-utilized VM instances when possible. Processes running on these VMs need to be migrated to other VMs, and most of the under-utilized VMs are deleted.

VI. Implementation

A. Proof of concept model

We have implemented a prototype of CSRM using processes running in linux containers (LXC) [15] on Oracle VirtualBox VMs. Containers offer isolated execution environments resembling the applications running on library OSes, and thus can be migrated between virtual machines unlike linux processes. Each of our workloads was run as a process in a separate container. These workloads consisted of SPEC CPU2006 benchmarks running in an infinite loop. The Checkpoint and Restore In User mode (CRIU) utility [9] was used for container migrations across VMs. CRIU checkpoints the state of running containers and saves it to disk.

We minimized the migrated state by storing the root filesystem of containers on each VM, and using the rsync utility to rapidly sync the container state. CRIU requires support from the linux kernel, for which we modified the configuration of linux kernel version 3.19.3 suitably. The Migrator and Observers are implemented as daemonized threads in C, which communicate via sockets. The Observer threads extract per-process CPU and memory demands from the /proc/ filesystem in Linux [16]. The Migrator enforced the migration policy described as per the Single process per VM case.

The aim of this proof of concept model was to demonstrate the robustness and utility of our CSRM architecture. However, on account of the instability of lxc container checkpoint and migration (container migration is not officially supported yet, while the limited checkpointing capability is offered in a developmental version of lxc), no meaningful long-term experiments were completed.

B. Simulator

The effectiveness of the CSRM is governed by the migration policy implemented on it. In order to rapidly test out various migration policies under different workload scenarios, we developed a simulator to mimic the behavior of the real system. The simulator allowed us to create robust policies by uncovering undesirable corner cases resulting from certain policy decisions. Policies are implemented as classes which communicate with the simulator through a simple API. The overall schematic of the simulator is shown in Figure 3. The simulator and the workload generator are implemented as Java applications in about 2000 lines of code.

Sample workloads are generated using a simple workload generator written by us. The workload generator takes the average long-term resource consumption of a process as input, the time duration of simulation as well as the behavior of the application (static, low-variability and high-variability) and generates a trace file containing the sample resource demands for each minute of its execution.

The simulator first sets up its VM cluster model based on the initial list of processes. Observer objects are created for each active VM which parse the relevant process trace files. This information is passed on to the Migrator object regularly, which analyzes the situation and implements the migration policy object instantiated. Each migration policy is implemented as a separate class, which uses the global state received from the Migrator object on each simulation tick and dictates the migration of VMs or the creation/removal of VMs. The simulator tracks the cumulative operating costs, number of migrations per process and also displays a detailed trace of all migration events for debugging.

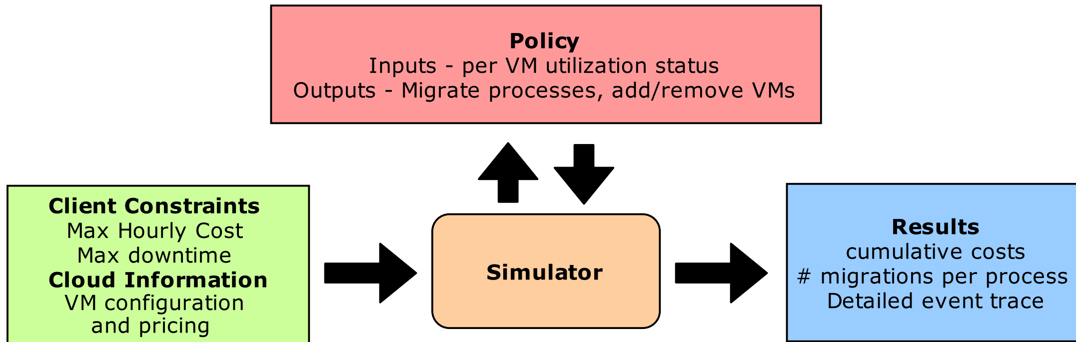

Fig. 3: Overall schematic of the simulator

VII. Evaluation

A. Migration Policies

We used our random workload generator to generate the resource consumptions for various workloads. Six different processes, two from each of the different resource usage patterns (Figure 1 and [10] were simulated for three days. These workloads had average memory footprints of about 800 MB. Observers sampled the resource consumption of applications once every minute. Initially, simulations were run by setting unrealistically high values for the total number of migrations per process and operating cost. This was done to identify the optimal values of these parameters. Subsequently, we studied the influence of constraining the policies along each of these metrics by supplying smaller inputs. The price and configurations of different VM instances are as per current AWS policy.

1) Cost Benefits: For the six workloads, based on the best-fit provisioning for the peak resource requirements, the total operating cost for a period of 3 days is $35.42, while both of the migration policies used up $26 for the same duration. This is a reduction of 26.5% for a limited simulation. The cost-efficiency of Multiple-Policy is expected to increase with the number of running processes, as it can better utilize bigger and more cost-effective VM instances. The cost-efficiency of both policies should increase with increasing dynamic variations in application requirements.

2) Capping Costs: Unbounded simulations reported that an hourly operating cost of about $5 is needed to optimize the utilization of reserved VMs, and allow all applications to perform at peak levels. Consequently, further simulations were run with decreasing operating budget to observe the effect of cost on migration and memory overcommitment. As explained in Section VII, memory overcommitment is representative of performance degradation of applications as well as VM utilization.

As seen in Figure 5, increasing the operating budget available minimizes the performance degradation of applications to a certain limit. Once memory overcommitment is below 100%, increasing the operating budget pushes the VM utilization closer to 100%. This is because while the spending limit is an upper limit, having a greater upper limit allows the Migrator greater possible configurations to optimize the system. This effect can be observed in Figure 4, as greater number of migrations are allowed to happen on increasing the cost barrier. However, this approach has diminishing returns closer to 100%. The uncharacteristic number of migrations for One-Policy with a budget of 4 is due to the fact that our migration policies prioritize the optimization of VM utilization.

The simpler One-Policy was 15-30% cheaper than Multiple-Policy, while delivering worse performance degradation and VM utilization of 10-38%.

3) Decreasing Downtime: Five migrations per process emerged as a sufficient limit to fully optimize the VM cluster for the given workloads. We ran further simulations while further constraining this limit to observe the degree of sub-optimal

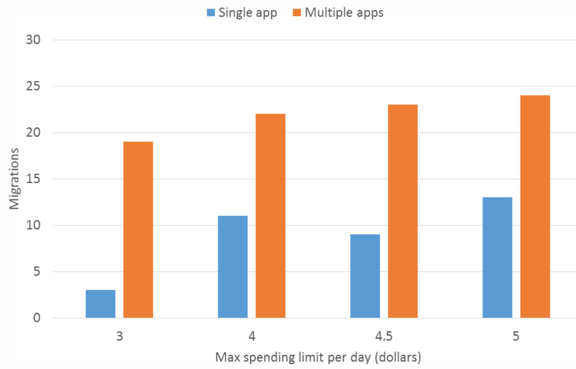

Fig. 4: Effect of capping costs on number of migrations

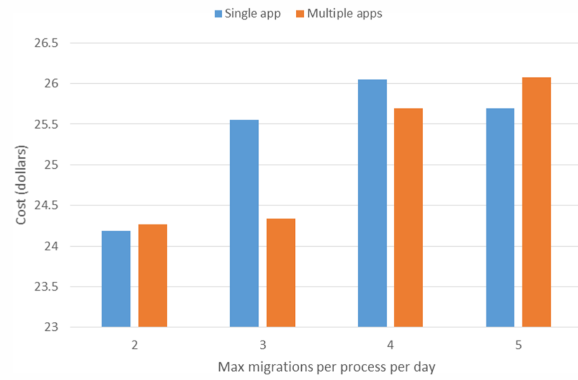

Fig. 6: Effect of limiting migrations on total costs

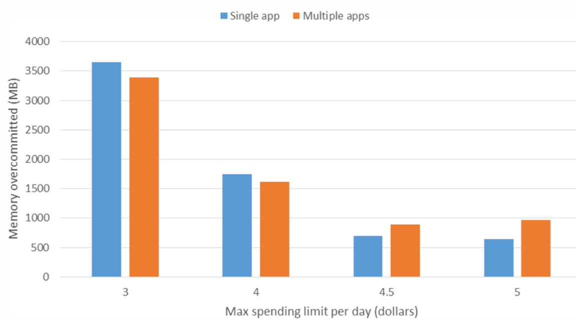

Fig. 5: Effect of capping costs on memory overcommitment

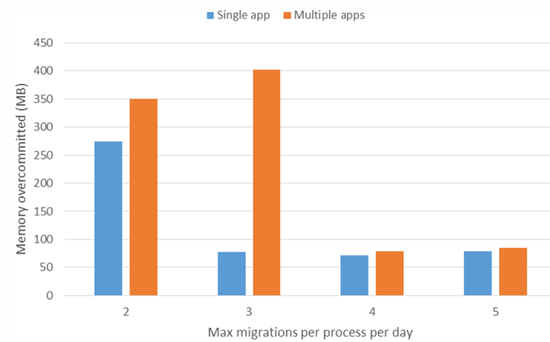

Fig. 7: Effect of limiting migrations on memory overcommitment

performance obtained on limiting this parameter.

Limiting the number of migrations allowed decreases the possible options available to the Migrator for improving the utilization of the system. This reduces the overall operating costs as seen in Figure 6 but hurts the performance of the system significantly as depicted in Figure 7. The high memory overcommitment for Multiple-Policy with 3 allowed migrations is due to the system moving into a state of local minima, without any further migrations available.

Comparing the two policies, the simple One-Policy remains immune to corner-cases as seen in the case of Multiple-Policy, while delivering almost comparable performance (above 90% of the performance of Multiple-Policy with 4 and 5 allowed migrations), while being only a maximum of 4.7% less cost-effective.

4) Suppressing Spikes: There are instances where the resource requirement of an application rises sharply, but only remains high for a short time. Such bursty behavior could trigger unnecessary application migrations as well as VMs creations. In order to ween out such spurious spikes, the Observers track a running median of the resource footprints of all applications. As a result, migrations are only triggered in response to sustained high or low activity.

We evaluated the effect of the window size for this tracking on the overall performance of the VM cluster. Figure 8 demonstrates how increasing the tracking window size from 1 to 8 intervals reduces the number of migrations by 10% for One-Policy, while not showing significant changes for Multiple-Policy. This is as One-Policy uses the Best-Fit algorithm for each application, so sudden spikes in application resource requirement trigger scale-up/down events. When multiple applications are housed on a single VM, larger VM configurations are instantiated which improve capacity to absorb changes in resource requirements. However, these are highly workload-specific scenarios.

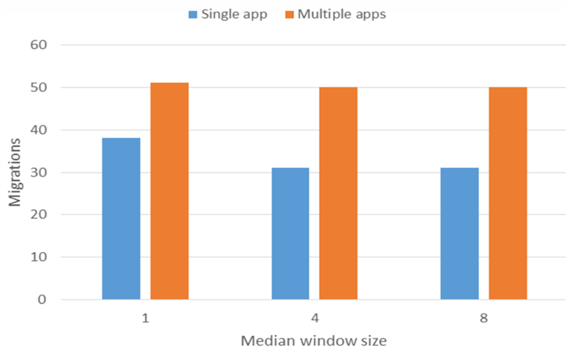

Fig. 8: Effect of tracking window size on migration count

VIII. Related Work

Ben-Yehuda et al. proposed Resource as a Service (RaaS) [2] to signal a shift from the current IaaS model of cloud computing. They advocate RaaS clouds, where clients can lease resources like RAM in terms of pages and CPU in terms of cycles. This reduces client costs by eliminating the charge for idle resources. However, the clients still have to ensure that their applications use resources diligently to avoid unnecessary costs. In addition, it is difficult for application developers to make accurate estimates of the resources required by their application which makes predicting costs and budgeting difficult. Our solution hides this complexity from the application developer. Also, RaaS requires a paradigm shift in the cloud computing model, while our approach works within the current IaaS model.

An approach closely related to our proposal is implemented in CloudOS [13] in the form of a middleware framework which uses application-level migration to optimize VM utilization. The framework consists of node managers which monitor the resource consumption on each VM, and a COS manager which has a global view and initiates migration decisions. A new actor-oriented programming language and runtime called SALSA is needed to build applications for CloudOS. The SALSA runtime provides support for actor migration, which is used to achieve the goal of optimizing VM utilization. This approach is suitable only for distributed applications, which need to be rewritten in the SALSA programming language. Also, the framework considers all VMs as identical, and hence does not incorporate cost efficiency or VM suitability in its migration policies.

Condor [17] is a scheduling system used to schedule multiple jobs in a large cluster of workstations interconnected by high capacity networks. It leverages the fact that workstations are underutilized by most users and schedules background jobs on them. On a similar note, Quasar [11] has been proposed as a cluster manager that maximizes the resource utilization of data centers. Both Condor and Quasar use migration to improve the utilization of static clusters and data centers, which draws some parallels to our problem. Implementing Quasar or Condor in the IaaS context is beneficial to cloud service providers, to accommodate the most VMs possible on the physical infrastructure and handle overcommitment through VM migration. However, our solution aims at improving cost efficiency for the cloud service user.

Client-side cost efficiency has also been tackled previously through optimal resource provisioning schemes. There are two popular resource provisioning strategies available in commercial clouds, viz. reserved and on-demand resources. Both proposals argue that neither of these two approaches results in optimal payment for the consumed resources. Delimitrou et al. [10] propose hybrid strategies which use a combination of reserved and on-demand resources to guarantee cost efficiency. However, the optimal strategy is decided via a priori profiling of incoming applications, and does not update dynamically with real-time behavior of applications.

On the other hand, the Optimal Cloud Resource Provisioning algorithm [8] uses a stochastic programming model to guarantee optimal resource provisioning. The algorithm handles dynamic changes in a program's behavior through regularly updated resource provisioning. However, the paper lists an inability to handle special cases like Valentine's Day, which drives up e-commerce demands as a limitation.

IX. Conclusion

As individuals and organizations increasingly look towards public clouds for their compute needs, it is becoming crucial to reason about the cost-efficiency and utilization of reserved resources in

the cloud. The flexible pricing plans currently offered by cloud providers like Amazon and Google aren't sufficient to allow users to streamline their cloud operations. Towards this end, we proposed CSRM in this work to give users greater control of their cloud investments through a combination of application mobility and real-time management.

We also evaluated two policies for the efficient organization of cloud applications on a VM cluster, with a view to increase utilization and satisfy performance guarantees while keeping costs within a budget. While optimal migration policies are almost impossible to develop, evaluation results show that simple heuristic-based policies are a good alternative. Our results show that such policies offer about 25% reduction in operating costs, which comes from improving the VM utilization.

ACKNOWLEDGMENT

The authors would like to thank the CS736 class of Spring 2015 for their useful feedback, and particularly Aditya Venkataraman, Nidhi Tyagi, Junhan Zhu and Prasanth Krishnan for their helpful reviews.

REFERENCES

[1] Stephen Adler. The slashdot effect- an analysis of three internet publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html. Accessed: 2015-05-01.

[2] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The rise of raas: the resource-as-a-service cloud. *Commun. ACM*, 2014.

[3] Amazon ec2 pricing. http://aws.amazon.com/ec2/pricing/. Accessed: 2015-05-01.

[4] Amazon web services is now a $5 billion business. http://www.theverge.com/2015/4/23/8485501/amazon-earnings-q1-2015-aws-web-services-5-billion. Accessed: 2015-05-01.

[5] T. E. Anderson. The case for application-specific operating systems. pages 92–94, 1992.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[7] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10.

[8] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE T. Services Computing*, 5(2):164–177, 2012.

[9] Checkpoint/restore in userspace. http://criu.org/Checkpoint/Restore. Accessed: 2015-05-01.

[10] Christina Delimitrou and Christos Kozyrakis. Optimizing resource provisioning in shared cloud systems. Technical Report CSTR 2014-06 11/25/2014, Stanford University, November 2014.

[11] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[13] Shigeru Imai, Thomas Chestna, and Carlos A. Varela. Elastic scalable cloud computing using application-level migration. In *IEEE Fifth International Conference on Utility and Cloud Computing, UCC 2012, Chicago, IL, USA, November 5-8, 2012*, pages 91–98, 2012.

[14] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv - optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 61–72, 2014.

[15] Linux containers (lxc). https://linuxcontainers.org/. Accessed: 2015-05-01.

[16] Linux filesystem hierarchy - proc. http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html. Accessed: 2015-05-01.

[17] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, USA, June 13-17, 1988*, pages 104–111, 1988.

[18] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2013*.

[19] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[20] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011*.

[21] B. Speitkamp and M. Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *Services Computing, IEEE Transactions on*, 3(4):266–278, Oct 2010.

[22] Carl A. Waldspurger. Memory resource management in vmware ESX server. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.