Continuous Datacenter Consolidation

Petter Svärd, Wubin Li, Eddie Wadbro, Johan Tordsson, and Erik Elmroth

Department of Computing Science, Umeå University
SE-901 87 Umeå, Sweden
Email: {petters, wubin.li, eddiew, tordsson, elmroth}@cs.umu.se

***Abstract*—Efficient mapping of Virtual Machines (VMs) onto physical servers is a key problem for cloud infrastructure providers as hardware utilization directly impacts revenue. Today, this mapping is commonly only performed when new VMs are created, but as VM workloads fluctuate and server availability varies, any initial mapping is bound to become suboptimal over time. We introduce a set of heuristic methods for continuous optimization of the VM-to-server mapping based on combinations of fundamental management actions, namely suspending and resuming physical machines, migrating VMs, and suspending and resuming VMs. Using these methods cloud infrastructure providers can continuously optimize their server resources regardless of the predictability of the workload. To verify that our approach is applicable in real-world scenarios, we build a proof-of-concept datacenter management system that implements the proposed algorithms. The feasibility of our approach is evaluated through a combination of simulations and real experiments where our system provisions a workload of benchmark applications. Our results indicate that the proposed algorithms are feasible, that the combined management approach achieves the best results, and that the VM suspend and resume mechanism has the largest impact.**

***Keywords*-Cloud Computing; Scheduling; Heuristic Methods; Consolidation; VM Migration; Power Management.**

I. Introduction

To date, most research on Virtual Machine (VM) provisioning for cloud datacenters has focused on deploy time scheduling, typically formulated as assignment problems where VMs are mapped to Physical Machines (PMs). Common objectives for these formulations are to optimize criteria such as Service Level Agreements (SLAs) [5], provider revenue [9], performance [23], utilization [37] etc. or a combination thereof. Notably, there are several factors that complicates this problem. First of all, VM scheduling is an online problem as both the arrival rate of new VM requests and the completion time for provisioned VMs is unknown. In addition, resource usage of individual VMs also varies over time. Changes to the server pool, due to failures or energy-management actions such as power-off and frequency-scaling, can also impact the performance of deployed VMs.

These factors imply that any scheduling solution may become suboptimal over time. To address this, we propose a continuous VM remapping approach to optimize VM provisioning as a complement to VM scheduling. Our approach consists of a set of algorithms that enable cloud infrastructure providers to automatically reconfigure the mapping of VMs to PMs and adapt to the changes in workloads and the physical environment. These algorithms are based on a combination of management actions, i.e., suspend and resume of PMs, live VM migration, and suspend and resume of VMs. This *continuous datacenter consolidation* approach aims to maximize cloud provider revenue over time by minimizing power consumption, maximizing PM utilization, and prioritizing important VM requests.

The remainder of the paper is organized as follows. Section II describes our system model and the assumptions made. Section III briefly elaborates on the problem and outlines our optimization algorithms. Section IV describes the architecture and implementation of our continuous datacenter consolidation engine. Section V presents the experimental evaluation on synthetic workload. Section VI surveys the related work on datacenter management, autonomic computing, and VM scheduling. Our conclusions are given in Section VII followed by a presentation of future work, acknowledgments, and a list of references.

II. System Model Overview

This work is based on a model where a datacenter consists of a set of PMs that are used to provision VMs on behalf of users. For each time unit the datacenter provisions a VM for a user, a profit r is generated, whereas a penalty f is imposed to the provider if the VM is not running for a time unit. In addition to any such SLAs, penalties, cloud providers pay electricity costs for running the PMs. To model the power consumption P of a physical server, as a function of the CPU utilization u, we adopt the following model presented by Blackburn [24]:

$$P(u) = P_{idle} + (P_{max} - P_{idle})u, \tag{1}$$

where P_{max} and P_{idle} are the power consumption for a server at full load and at idle respectively. In this model, server power consumption scales linearly with CPU utilization. So for example, a server with $P_{idle} = 200$ W, $P_{max} = 300$ W, and 10% CPU utilization requires $P(10\%) = 200\text{ W} + (300\text{ W} - 200\text{ W}) \times 10\% = 210$ W. Through empirical measurement of various servers, this approximation has proven by Baroso et al. to be accurate to within $\pm 5\%$ across all CPU utilization rates [4]. They also argue that although the power consumption of the CPU only accounts for roughly 40% of the power usage of a server, it can be used to model the total usage [4].

Finally, the power consumption of the whole data center can be expressed as

$$P_{tot} = \sum_{i|\text{physical machine } pm_i \text{ is active}} P(u_i),$$

where u_i denotes the CPU utilization of physical machine pm_i. The monetary cost of running the data center during a time interval $(0, t)$ is therefore given by

$$C = \frac{m}{1000} \int_0^t P_{tot},$$

where m represents the unit cost per kWh.

When mapping VMs to PMs it is possible to provision more virtual resources than what is physically available in the PM, a concept known as overbooking [33]. However, some infrastructure providers e.g., Amazon EC2, provide a one-to-one mapping between virtual and physical CPU and memory resources [10], which means that PMs are not being overbooked. Within the context of this work, we assume that no overbooking is taking place.

Research on proactive optimization based on the historical data and workload prediction has been performed extensively, by e.g. Ali-Eldin et al. [2]. In our model, we assume that the infrastructure provider has no knowledge of the future workload and no prediction is taking place. However, we believe that it is possible to enhance our approach with such prediction techniques.

A core technology to optimize the mapping of VMs to PMs during operation of the datacenter is live VMs migration [6]. Live VM migration allows a running VM to be moved from one PM to another without shutting it down. Live migration thus reduces SLA penalties as the VM is accessible by users during migration, but at the cost of migration taking considerable time to complete and consuming much hardware resources. To reduce both migration time and resource usage, we use the KVM XBZRLE delta compression migration algorithm [32] in our work as delta compression can significantly reduce the migration downtime, migration time and the amount of transmitted data during live migration for memory-intensive workloads [32], [36]. The total migration time of a VM is given by

$$dt = t_i + t_c + t_s + t_r,$$

where t_i, t_c, t_s, and t_r denote the time for iteratively transferring memory pages, the time for suspending the VM at source, time for CPU/BIOS transfer and the time for resuming the VM at destination, and pulling respectively. The three latter operations are usually fast and vary little between VMs, that is, t_c, t_s, and t_r are typically small. The iterative transfer time is harder to predict, but is usually a function of the active memory usage and the memory size of the VM and the network bandwidth b.

III. The proposed approach and heuristic methods

At any point in time, multiple events can take place. We define three events and prioritize them in descending order as *PM crash, VM exit*, and *VM arrival*. Our proposed approach dynamically handle these events according to their priorities, and adapts the cloud infrastructure to the environmental changes in a reactive manner. Simple management actions are used for optimization of the datacenter, i.e., (i) suspend/resume VMs, (ii) VM migration, and (iii) suspend/resume PMs. For an event of PM crash, a crashed PM not only affects all VMs hosted, but it must also be excluded as a potential destination for VMs. Upon an event of PM crash, we simply suspend all VMs hosted on that PM. A VM exit event has a higher priority than a VM arrival event, as capacity released by a terminated VM can be used to accept more VMs into the datacenter. When a VM terminates, the occupied resources are released, increasing the residual capacity of a PM. All VMs arriving are added to a list, namely, $candidateList$, which also may include VMs suspended in the past. VMs in $candidateList$ can be prospectively executed depending on the decision by the optimization process. A summary of the actions taken on the occurrence of events is shown in Algorithm 1.

Algorithm 1: handleEvents($events$)

```
for e ∈ events do
    if PM pm crash then
        Exclude pm and all VMs hosted;
    else if VM vm quits then
        Release capacity occupied by vm;
    else if VM vm arrives then
        Add vm to candidateList;
```

Once the event handling procedure is completed, a consolidation action presented in Algorithm 2 is triggered to optimize the profit gained by VM provision. In particular, if an infrastructure provider has too limited capacity, the profit can be maximized by selecting which VMs to run. In order to make an optimal selection, the following two questions need to be answered.

1. **Which VM should be placed first?**
   Intuitively, given a set of VMs, the ones that are most profitable should be placed with higher priorities. However, in our model, we also need to consider the penalty of suspending a VM. In this contribution, we prioritize a VM using the sum of its associate profit and penalty. For example, given two VMs, vm_1 with profit r_1 and penalty f_1, and vm_2 with profit r_2 and penalty f_2, our algorithms place vm_1 prior to vm_2 if $r_1 - f_2 > r_2 - f_1$ (even when $r_2 > r_1$).

TABLE I
EXAMPLE OF VM PRIORITIZATION.

Option	Profit	Penalty	Gain
Run vm_1, and suspend vm_2	r_1	f_2	$r_1 - f_2$
Run vm_2, and suspend vm_1	r_2	f_1	$r_2 - f_1$

Table I compares two different options with respect to potential gains from the perspective of the infrastructure

provider. In this case, the first option is preferable if $r_1 - f_2 > r_2 - f_1$, or equivalently $r_1 + f_1 > r_2 + f_2$.

2. **Which VM should be selected to be replaced?** Given a VM vm (with profit r and penalty f) that can not be hosted by any PM, it is possible to suspend another already running VM vm' (with profit r' and penalty f') in PM pm and instead run vm if (i) pm is capable of running vm (after the suspension of vm') and (ii) it is more profitable to run vm than vm'. Following the strategy aforementioned, the VM with minimum $(r'+f')$ is selected as the victim VM, i.e., the VM to be suspended.

Algorithm 2: consolidation()

```
   /* consolidation                                   */
1  if suspend/resume VM is allowed then
2  |  Add all suspended VMs to candidateList;
3  Sort VMs in candidateList by (price + penalty) in
   descending order;
4  for vm ∈ candidateList do
5  |  handleVM(vm);
6  if suspend/resume PM is allowed then
7  |  if VM migration is allowed then
   |  |  // Release PMs via VM migration.
8  |  |  releasePMsbyMigration();
9  |  else
   |  |  // Suspend PMs without VM running.
10 |  |  suspendIdlePMs();
```

In order to optimize the datacenter operation, our approach is to generate a list of *consolidation* actions according to Algorithm 2. If the used algorithm allows for suspend/resume of VMs, the first step is to recycle all the currently suspended VMs and enable them to be possibly resumed by adding them to $candidateList$ (see Line 2). All VMs (also referred to as *object* VMs) in $candidateList$ are to be handled sequentially, in a descending order that they are ranked by $(price + penalty)$. There are two possible outcomes of the action **handleVM**, i.e., either suspend the current VM, or place and start VM in some physical server. The final step in a round of optimization is to suspend idle PMs if the feature is enabled (see Line 6–10). More PMs may be released and then suspended, depending on whether VM migration is allowed.

As depicted in Line 2 in Algorithm 3, we use *best-fit* as the baseline strategy to find an active PM for a VM. The motivation for this is to load each PM as much as possible, maximizing the utilization of the PMs and thus minimizing the residual capacity of the whole infrastructure. If this is not feasible (i.e., no PM can host the VM), a simple solution is to try starting a suspended (or a new) PM (see Line 6) and place the VM there. However, in order to decrease the total number of active PMs, prior to starting a new PM, the proposed algorithm strives to readjust the placement of VMs, to see if there exists a PM that can host the VM after migrating some VMs to other PMs (see Line 4). The details of this can be found in Algorithm 4).

Algorithm 3: handleVM(vm)

```
   /* This function is for handling newly
      arrival VMs and VMs that were
      suspended in the previous period.  */
1  pms ← active PMs;
   // Find a PM for vm using the best-fit
      strategy.
2  pm ← best-fit(vm, pms);
3  if pm not found and VM migration is allowed then
4  |  pm ← findPMbyMigration(vm);
5  if pm not found then
6  |  pm ← attempt to start a new PM;
7  if pm not found and suspend/resume VM is allowed then
   |  // Find a victim VM and replace it
   |     with vm.
8  |  pm ← findPMwithVictimVM(vm);
9  if pm found then
10 |  placeVM(vm, pm);
11 else
12 |  suspendVM(vm);
```

Finally, if no suitable PM is found after trying all of the above approaches, an aggressive approach is applied to pick one of the running VMs as the victim, suspend it, and replace it with the object VM (see Line 8, as described in Algorithm 5). This step is conducted only if replacing the victim with the object VM is feasible and more profitable. Note that our algorithm currently only selects one VM as victim, it is however possible to extend this to enable selection of multiple VMs as victims instead.

To find if re-arranging the mapping of VMs by live migration them can make room for vm on some PM, all active PMs are sorted by residual capacity in descending order in Algorithm 4. The PMs are then evaluated by looking at the feasibility of migrating a set of VMs to other PMs. When evaluating a PM, we only consider migrating VMs that are smaller than vm (see Line 6), as testing a VM larger than vm is meaningless (namely, if a PM can be found for this case, just place vm there without adjusting any VM placement). Also, note that as a machine can be represented by multiple dimensions (CPU, memory, storage, etc), the *size* function in Algorithm 4 can have different definitions depending on the application scenarios. In this work, it is defined in terms of CPU cores while other dimensions (memory, storage, etc.) are used as constraints when evaluating the feasibility of placement on PMs. The algorithm also strives to minimize the number of migrated VMs, as migration takes time and consumes resources. In addition, to further reduce the number of VMs migrated, all potential VMs are sorted by size in descending order (see Line 7). VMs to be migrated are added to a *plan* by function **addToMigratitonPlan** (see Line 13). The evaluation procedure stops when the first suitable PM is found, and the migration plan is executed by **commitMigratitonPlan** (see Line 20). If a PM is not suitable, the migration

Algorithm 4: findPMbyMigration(vm)

```
   /* Find a PM that can host vm after
      migrating some VMs to other PMs.   */
1  pms ← all active PMs;
2  Sort pms by residual capacity in descending order;
3  for p ∈ pms do
4      feasible ← FALSE;
5      vmSet ← VMs hosted in p;
6      vms ← {v ∈ vmSet | size(v) < size(vm)};
7      Sort vms by capacity in descending order;
8      pmset ← pms \ {p};
9      for v ∈ vms do
           // Find a PM (not p) to host v
              using the best-fit strategy.
10         pm ← best-fit(v, pmset);
11         if pm not found then
12             break;
13         addToMigratitonPlan(v, pm);
14         if p can host vm then
15             feasible ← TRUE;
16             break;
17     if not feasible then
18         cancelMigratitonPlan();
19         continue;
20     commitMigratitonPlan();
21     return p;
```

Algorithm 5: findPMwithVictimVM(vm)

```
   /* Find a PM that can host vm after
      suspending a vm hosted.            */
1  destination ← null;
2  minRF ← price(vm) + penalty(vm);
3  pms ← all active PMs;
4  for p ∈ pms do
5      vmSet ← VMs hosted in p;
6      vms ← {v ∈ vmSet | price(v) + penalty(v) <
       minRF};
7      Sort vms by (price + penalty) in ascending order;
8      for v ∈ vms do
9          if p can host vm after suspending v then
10             victim ← v;
11             destination ← p;
12             minRF ← price(v) + penalty(v);
13             break;
14 if destination is not null then
15     suspendVM(victim);
16 return destination;
```

plan is canceled by **cancelMigratitonPlan** (see Line 18).

Algorithm 5 introduces the strategy of finding a victim VM to be replaced by a VM in $candidateList$. The basic idea is to select the VM with the minimum value of $(price + penalty)$ among all selectable VMs (see Line 9-13). Once a VM is selected as a victim VM, it is suspended and moved to a waiting list and may potentially be resumed depending on the future optimization decision.

The final action in each consolidation process is to try to reduce the power consumption of the infrastructure by suspending all idle PMs, or by releasing more PMs by VM migration. To empty an active PM, all hosted VMs need to be migrated to other PM(s). Intuitively, PMs with higher residual capacity are more likely able to be emptied, and thus PMs are evaluated in the order of residual capacity (see Line 2 in Algorithm 6). Once again, we use a *best-fit* strategy whenever finding a new location for a VM (see Line 8).

Finally, by enabling or disabling the three management actions identified, we end up with 8 algorithms to evaluate in Section V, as listed in Table II.

Notably, scheduling algorithms based on bin-packing [7] or knapsack approaches are exponential in complexity, limiting their applicability for large-scale problems. As our algorithms are all with polynomial complexity this means that they can be used for larger problem sizes than what is studied in our simulations.

Algorithm 6: releasePMsbyMigration()

```
   /* Release PMs through VM migration   */
1  pms ← all active PMs;
2  Sort pms by residual capacity in descending order;
3  for p ∈ pms do
4      feasible ← TRUE;
5      vms ← VMs hosted in p;
6      pmset ← pms \ {p};
7      for vm ∈ vms do
           // Find a PM (not p) to host vm
              using the best-fit strategy.
8          pm ← best-fit(vm, pmset);
9          if pm not found then
10             feasible ← FALSE;
11             break;
12         addToMigratitonPlan(vm, pm);
13     if not feasible then
14         cancelMigratitonPlan();
15         continue;
16     commitMigratitonPlan();
       // Suspend PM p when it is idle.
17     suspendPM(p);
```

TABLE II
ALGORITHMS AND MANAGEMENT DIMENSIONS

algorithm	suspend/resume PMs	suspend/resume VMs	VM migration
baseline			
pmSR	✓		
vmSR		✓	
vmM			✓
vmSRpmSR	✓	✓	
vmMpmSR	✓		✓
vmMvmSR		✓	✓
combined	✓	✓	✓

IV. ARCHITECTURE AND IMPLEMENTATION.

To verify that our algorithms are valid in real-world scenarios, we design and implement a software package, Automatic Continuous Datacenter Consolidation (ACDC), capable of managing PMs and VMs using the algorithms described in Section III. PMs can be suspended and resumed, and VMs can be started, shutdown and migrated. The ACDC is built on the open-source KVM [18] hypervisor in combination with libvirt [22] to provide the virtualization backend.

The architecture consists of three components, as shown in Figure 1. All components are implemented in Java and the complete package runs on one PM, the controller. To perform consolidation actions on the worker PMs, where ACDC is not running, ssh remote invocation by means of shared key authentication is used. This means that remote libvirt virsh [22] commands for suspending and resuming VMs and migration of VMs, as well as scripts to suspend and resume PMs, can be executed by the controller PM.

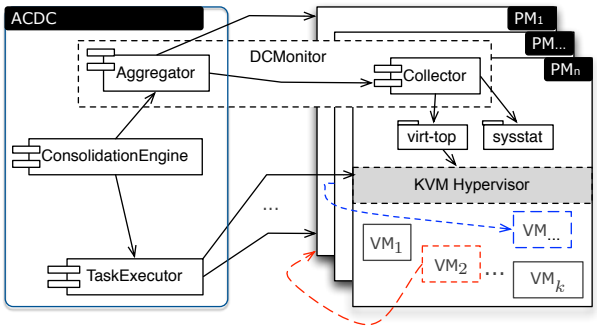

Fig. 1. Architecture overview.

A. DCMonitor.

To continuously collect the state information of the data center, we implement a monitoring system that consists of two subcomponents. A *Collector* installed in each physical machine, and an *Aggregator* is co-located with the ConsolidationEngine. The collector is built on two libraries, virt-top [13] and sysstat [1]. virt-top is a top[1]-like utility for showing stats of virtualized domains. It is employed to collect the cpu, memory and IO usage info of virtual machines. Note that virt-top only collects the total memory allocated to the guest, not the memory being used. The sysstat utilities are a collection of performance monitoring tools for Linux hosts. They are used to gather the resource usage info of physical hosts. Using the statistics collected by the Collector in each physical host, the Aggregator constructs a global view of the state of the data center.

[1]*top* is a task manager program inherent in many Unix-like operating systems.

B. ConsolidationEngine.

The ConsolidationEngine analyzes the data from the DCMonitor and decides on what actions to take in order to optimize the operation of the datacenter, according to the selected algorithm. All algorithms in Table II are evaluated through simulation. In addition to this, the combined algorithm and the baseline algorithm are also used to verify the simulation results for small-scale real experiments. The output from the ConsolidationEngine is an Execution Plan which is an ordered list of optimization actions.

C. TaskExecutor.

The TaskExecutor is designed to perform the actions produced by the ConsolidationEngine. It is running as a daemon, waiting to accept Execution Plans from the ConsolidationEngine. Once an Execution Plan arrives, it is processed and the tasks are performed in order. The TaskExecutor controls the underlying virtualized infrastructure by using libvirt virsh commands.

V. EVALUATION

We compare the eight algorithms defined in Table II by comparing the impact of each management action in isolation and combination on provider profit, PM utilization, number of running and suspended VMs. This is done both through simulations and real experiments. Note that, in order to increase the readability of the figures, data values are aggregated for each hour, unless otherwise specified.

A. Overall experiment setup.

We model the datacenter as a set of PMs where each server has 32 cores and 56G of memory. A limitation factor ($\chi = 0.9$) is set to restrict the maximum number of cores loaded for each PM, i.e., for each PM, 28 cores are allowed to be occupied by VMs. This is reasonable as some resources are needed by the hypervisor, for example to emulate the underlying hardware environment and to migrate VMs, and by the host operating system. The server power consumption is 100 W when idle and 560 W when fully utilized, which is consistent with power usage for the HP ProLiant DL165G7 servers used in the real tests. The price of electricity is set to be $0.07 per kW/h.

We consider 7 different VM instance types similar to offerings by Amazon EC2. Their hardware characteristics are illustrated in Figure 2, and their hourly prices as well as the associated SLA penalties for downtime are listed in Table III. VM arrival is modeled using one Poisson process for each instance type, seven processes in all. A parameter associated

with these processes is λ, which indicates the average arrival interval. Note that the time unit used in simulation is hours, while minutes are used in real test.

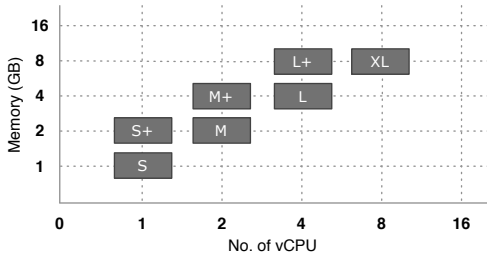

Fig. 2. VM instance types.

TABLE III
SETTING OF PRICES AND PENALTIES FOR VM INSTANCES.

VM types	S	S+	M	M+	L	L+	XL
Price ($/h)	0.04	0.06	0.09	0.14	0.20	0.31	0.46
Penalty ($/h)	0.40	0.60	0.90	1.40	2.00	3.10	4.60

B. Simulations.

For the simulations, we use 48 PMs, which is a common size for a rack in a datacenter [4]. We study three different scenarios with $\lambda = 0.4$ (high load scenario), $\lambda = 0.5$ (medium load scenario), and $\lambda = 0.6$ (low load scenario), respectively. Each new VM instance is assigned a lifetime ranging from 1 hour to 60 hours, uniformly distributed. By aggregating the capacity over all cores for each instance type (22 cores), we arrive at an average capacity requirement of $\frac{22}{\lambda} \times \frac{1+60}{2} = 1342$ cores per hour for the medium load scenario. These parameters are selected to make the average workload demand ($1342/(48*32) = 87.4\%$) close to the selected infrastructure limitation factor ($\chi = 0.9$). All VMs arrive during the first 252 hours, and then terminate within 60 hours after its arrival.

To compare the performances of algorithms on increasing the profit for the infrastructure provider, we run 10 tests for each value of λ. Note that the workload in each test case is the same for all eight algorithms. However, the workloads for any two tests are different, although they have the same VM arrival interval parameter.

1) Simulation Results: In the following sections we use one of the 10 medium load tests ($\lambda = 0.5$) as an example to investigate the behavior of the algorithms in detail. We study the profit (p') for each of the seven other algorithms compared with the profit (p) achieved by the baseline algorithm using a metric defined by $\alpha = (p' - p)/p$. The average α values of the 10 medium load tests are presented in Figure 3. Some interesting findings can be observed in Figure 3, including (i) that algorithms with suspend/resume of VMs enabled show more significant improvements with average α values higher than 20%, (ii) enabling suspend/resume PMs increase profit only slightly, by less than 5%, and (iii) using live VM migration can improve the profit by more than 10%.

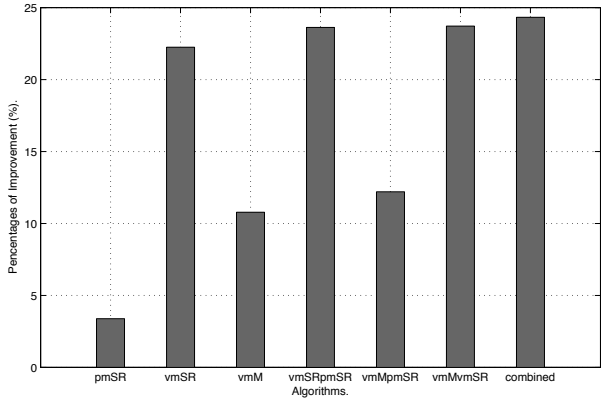

Fig. 3. Average profit improvement (all other algorithms vs. baseline).

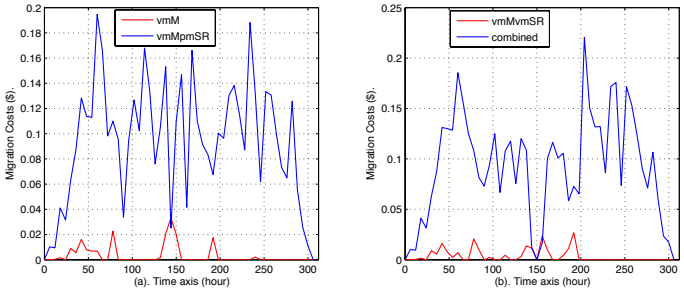

Fig. 7. VM migration cost: with/without suspend/resume PMs.

We next study the impact of the suspend/resume PM management action in more detail. Figure 4 shows that the average PM CPU usage with suspend/resume of PMs enabled is higher than without. In particular after 252 hours, these differences become larger, as there are no arriving VM requests resulting in a larger number of PMs that can be suspended. This is evident when VM migration is enabled (see subgraphs (c) and (d) in Figure 4, and subgraphs (c) and (d) in Figure 5). Enabling suspend/resume of PMs is also beneficial for the power consumption of the whole infrastructure. Similar to the CPU usage and number of active PMs, the differences are larger when the PMs are under low load compared with when the load is high, as illustrated in Figure 6. Also as expected, the difference for PM expenses is even larger when VM migration is enabled (see subgraphs (c) and (d) Figure 6);

Figure 7 demonstrates the collected average migration cost over time for algorithms with VM migration enabled. In order to improve the readability the migration costs are aggregated every 2 hours. Even so, we see that the migration costs are very low due to the short migration times. For example, it only takes 8 seconds to migrate an XL instance, resulting in a monetary cost of $0.0102. From Figure 7, we also observe that, for algorithms with suspend/resume PMs, VM migration costs are much higher than others, as Algorithm 6 migrates all VMs before suspending an active PM. Looking at subgraphs (c) and (d) in Figure 5, it is observed that benefiting from VM migration, more PMs can be suspended compared with other algorithms without VM migration.

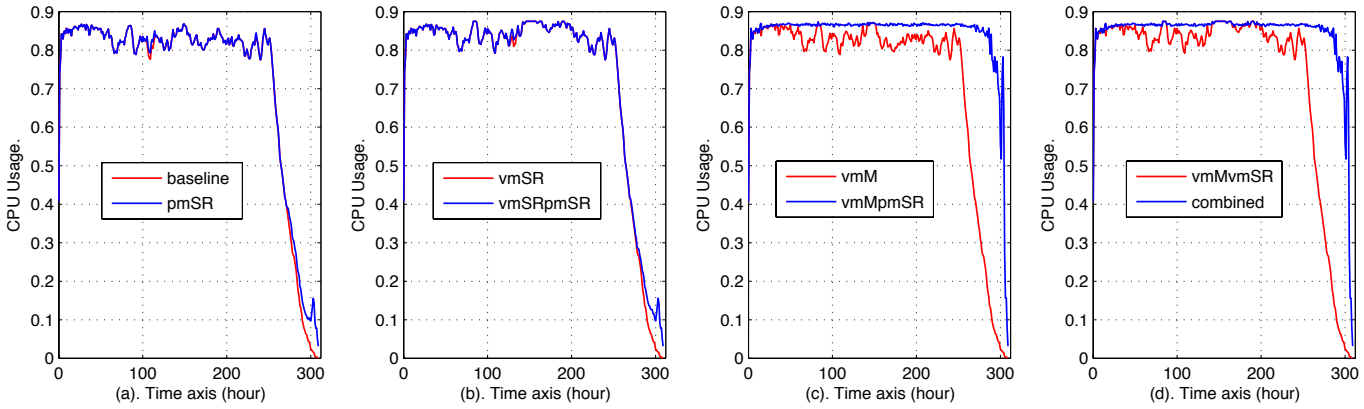

Fig. 4. CPU usage: with/without suspend/resume PMs.

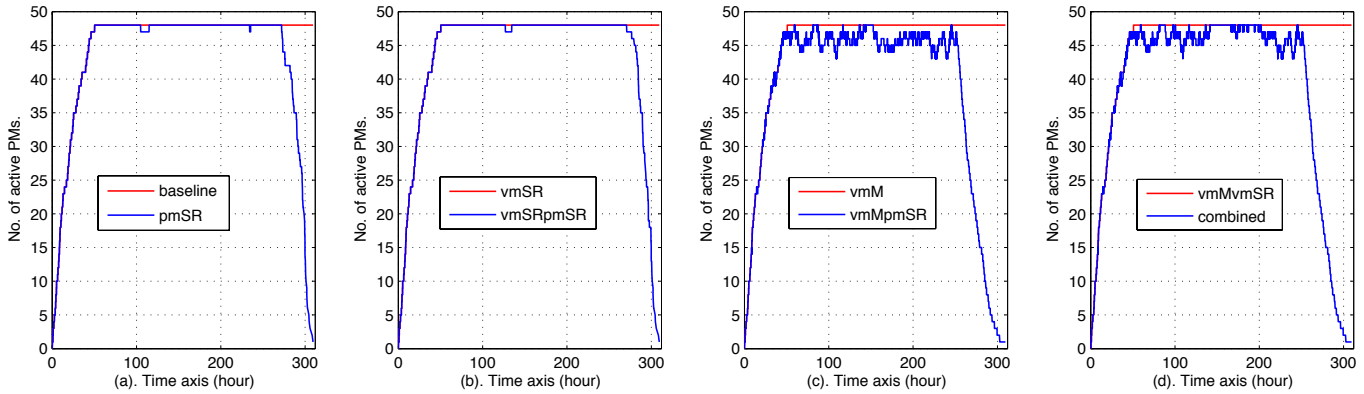

Fig. 5. No. of active PMs: with/without suspend/resume PMs.

Regarding the impact of VM migration, Figure 8 illustrates, for each policy, how CPU usage changes when migration is used. The impact is largest for the policies that allow suspend/resume of VMs (vmM)pmSR, and for the combined policy (as compared to vmSRpmSR). Looking at subgraph (c), we can see that there is almost no difference between vmSR and vmMvmSR. This is because in our current settings, suspend/resume VM is much more profit-efficient than VM migration, and it is dominating. This is also consistent with the data plot in Figure 3, where the differences among vmSR, vmSRpmSR, vmMvmSR, and combined are very small.

Turning to the impact of suspending and resuming VMs, Figure 9 plots the number of VMs suspended over time, as well as the their total number of cores. The subgraph (a) shows that, in general, the algorithms without the feature of resuming VMs (e.g., pmSR, and vmMpmSR) suspend fewer VMs than others. In particular, looking at the combined algorithm, we can see that it suspends the largest number of VMs most of the time (see the curve in bold). However, the total number of cores that belongs to suspended VMs is usually fewer than other algorithms (see subgraph (b) in Figure 9).

This observation indicates that VMs suspended by algorithms with this feature enabled are comparably smaller instances. This is consistent with our algorithm design, as when the capacity of the infrastructure is tight, Algorithm 5 selects VMs with smaller values of $(price + penalty)$ to suspend in order to release more capacity and run VMs with higher $(price + penalty)$ values. According to the settings in Figure 2 and Table III, smaller instances are using smaller $(price + penalty)$ values (but this is not a necessity). This is also consistent with the penalty plots in Figure 10.

Finally, we study the impact of the data center workload by varying the datacenter workload parameter (λ). Table IV presents aggregated results of 10 runs each with parameters $\lambda = 0.4$ (high load), $\lambda = 0.5$ (medium load), and $\lambda = 0.6$ (low load), respectively. In the high load scenario, the number of active PMs is high along with CPU usage. As there are quite a few VMs not running, the datacenter pays significant penalties and the profits are negative. Here, we note that baseline, pmSR, vmM, and vmMpmSR all perform very similar, with an average loss around $\$50$ per hour. In contrast, the algorithms that can start and resume VMs (vmSR, vmMvmSR, vmSRpmSR, and combined) and thus replace low-profit VMs when more important VM workload arrive, all perform much better with average losses around $\$11$ per hour. In the medium load case, we observe similar resource usage in the high load scenario, but for medium load, there are only a few VMs that are suspended as the average penalties are much lower and the

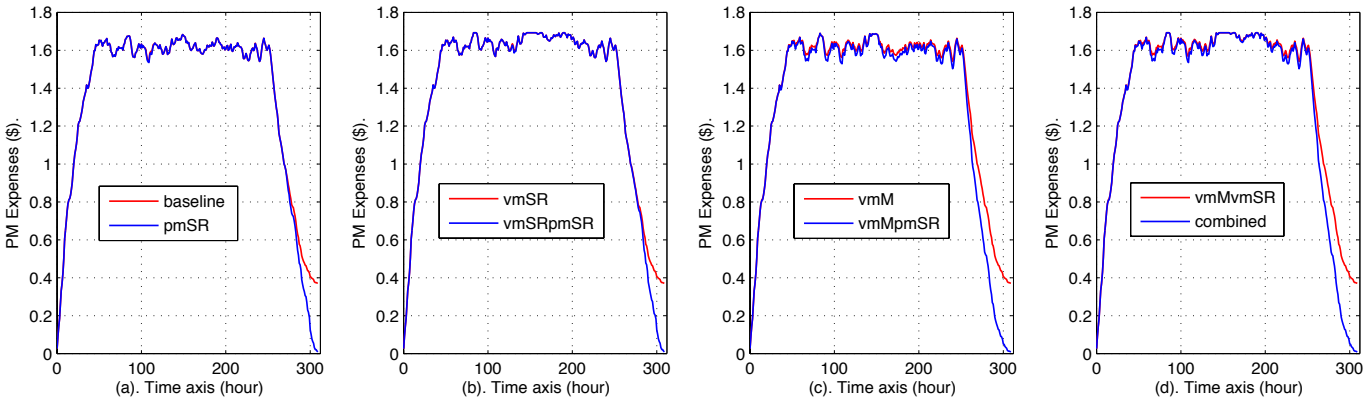

Fig. 6. PM expenses: with/without suspend/resume PMs.

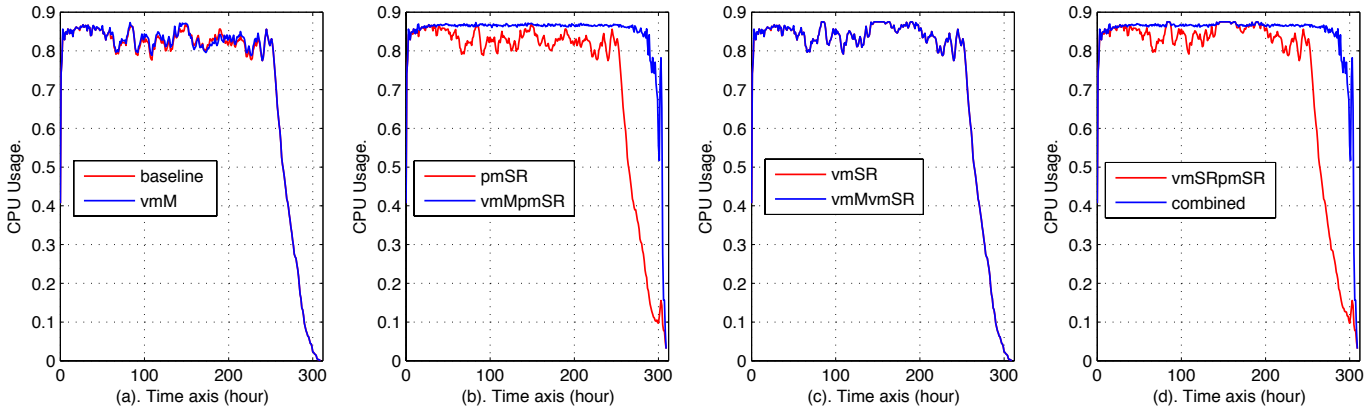

Fig. 8. CPU usage: with/without VM migration.

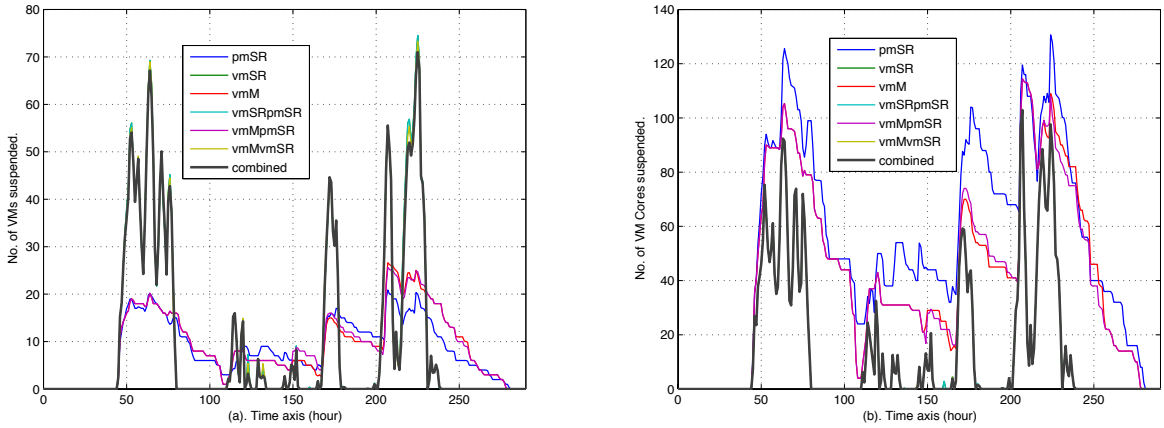

Fig. 9. No. of VMs suspended and No. of VM cores suspended.

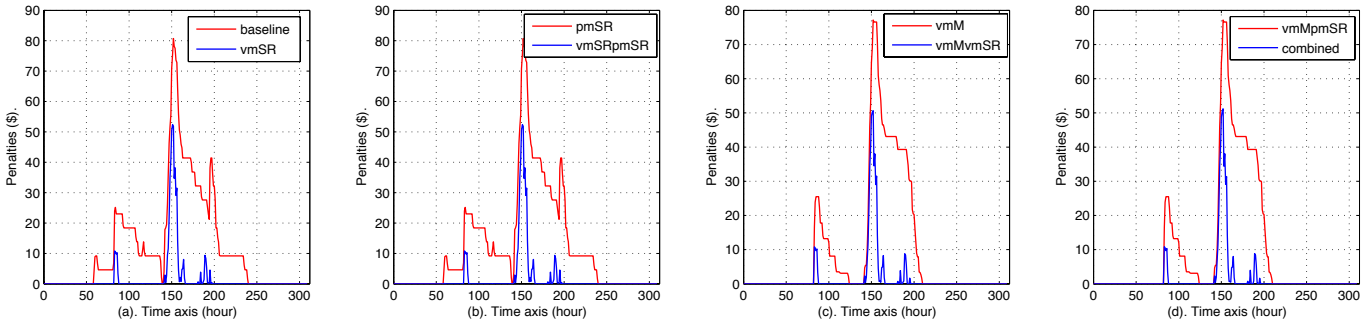

Fig. 10. Penalties with/without suspend/resume VMs.

TABLE IV
AGGREGATED RESULTS FOR DIFFERENT WORKLOADS.

λ	metrics	baseline	pmSR	vmSR	vmM	vmSRpmSR	vmMpmSR	vmMvmSR	combined
0.4	profit ($/h)	-56.15	-56.13	-11.33	-48.60	-11.31	-49.73	-11.40	-11.30
	penalty ($/h)	118.20	118.20	77.42	111.32	77.42	112.39	77.48	77.42
	no. act. PMs	45.7	43.3	45.7	45.7	43.2	39.4	45.7	39.2
	CPU usage (%)	74.20	74.71	76.74	75.07	77.29	84.36	76.75	84.94
0.5	profit ($/h)	48.10	49.72	58.80	53.28	59.46	53.96	59.50	59.80
	penalty ($/h)	12.05	12.05	1.72	9.67	1.72	9.67	1.66	1.61
	no. act. PMs	45.1	43.1	45.0	45.1	42.6	38.8	45.1	38.6
	CPU usage (%)	72.73	73.47	73.85	73.00	74.53	84.24	73.87	85.31
0.6	profit ($/h)	50.66	50.68	50.66	50.66	50.68	50.71	50.66	50.71
	penalty ($/h)	0	0	0	0	0	0	0	0
	no. act PMs	39.7	36.6	39.7	39.7	36.7	31.8	39.7	31.8
	CPU usage (%)	69.88	71.66	69.88	69.89	71.66	83.72	69.89	83.72

datacenter is profitable. Regarding the different algorithms, we note that also here, the ability to suspend and resume VMs is the key, with these four algorithms keeping penalties around $1.7 per hours and profits around $59 per hour. Notably, as the PMs are less loaded in this scenario, migration of VMs actually make a difference, with vmM and vmMpmSR having penalties of $9.7 as compared to $12 for baseline and pmSR. In the low load scenario, no penalties are paid as there for all algorithms always are enough resources to run all VMs. There are some differences in average number of PMs used and subsequently in CPU usage. The ability to suspend and resume PMs brings the average number of PMs down from 39.7 (algorithms baseline, vmSR, vmM, and vmMvmSR) to 36.7 (pmSR) and 36.7 (vmSRpmSR). Combining this with VM migration to be able to achieve consolidation has even greater impact, with 31.8 PMs used on average for vmMpmSR and the combined algorithm.

C. Real-world Demonstration.

In order to verify the validity of our approach, we perform a real-world test on a small testbed with 5 nodes, using our software described in Section IV. The PMs used are HP ProLiant DL165G7 @ 2.1 GHz with 32 cores and 56 GB of RAM each, connected by a top-of-the-rack Gigabit Ehternet switch. One node is functioning as controller and the other four nodes are worker nodes, hosting VMs. The setup of the testbed in terms of nodes and installed software is shown in Figure 11.

One debatable point is the setting of the peer-to-peer connection bandwidth among PMs, as network congestion issues can reduce the available bandwidth. However, we argue that by tuning the over-subscription factor and constructing the network topology in a proper manner [4] the effect of this problem can be reduced. Also, as VM migrations are carried out in a sequential order using our approach, network congestion potentially introduced by parallel VM migration can be mitigated.

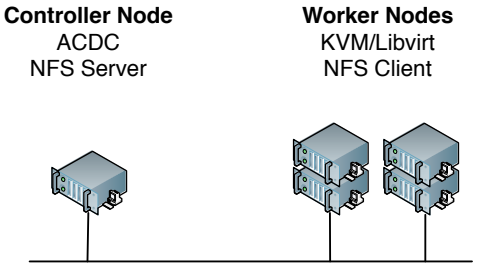

Fig. 11. Setup of the testbed.

With a limitation factor $\chi = 0.9$, the maximum number of cores available for VM provisioning is $4 * 32 * 0.9 = 115$. The test starts with an empty datacenter with new VMs arriving at an average rate of 6 minutes per instance type, namely, following a Poisson process with $\lambda = 0.1$. The VM lifetime is set to $24 - 36$ minutes normally distributed and the number of VMs to be provisioned during 96 minutes is 69. All VMs arrive in the first 60 minutes. For the real-world demonstration

we use the same instance types as in the simulation and the number of instances of each type is limited to 10. Using this configuration, the average demand of cores during the test duration of 96 minutes is 105 which is close to the maximum 115.

Each VM is running a synthetic benchmark *bw_mem*, which is a memory write benchmark from the LMBench [25] suite. The bw_mem benchmark allocates twice the specified amount of memory, zeros it, and then copies the first half to the second half. For each of the instance types, the amount of memory allocated to bw_mem and the number of parallel threads used is tuned in order to consume 50% of the VMs memory and 100% of the vCPU.

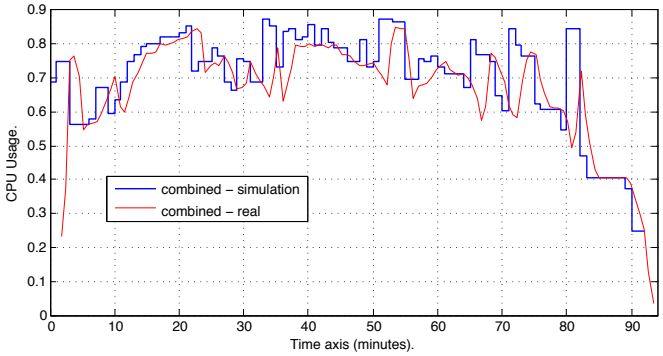

Fig. 12. CPU usage with *combined*: simulation vs. real.

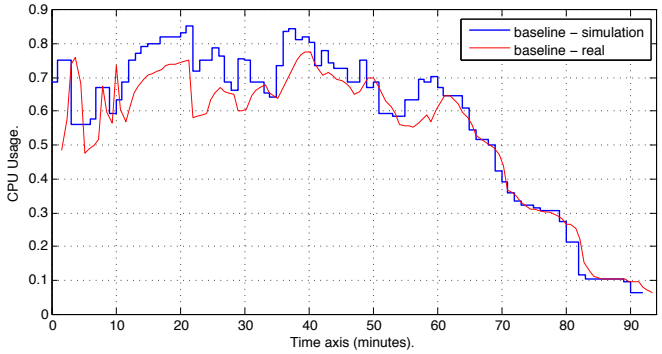

Fig. 13. CPU usage with *baseline*: simulation vs. real.

Figure 12 and Figure 13 present the CPU usage of the whole testbed over time in simulation and real tests. First of all, we remark that the behaviour of our algorithms (combined and baseline) in simulation is consistent with that in real tests. Another observation is that, in most cases the CPU usage in the real-world test is lower than in the simulation. The reason for this is that, in the simulation the CPU usage of a PM is defined by the proportion of cores occupied by VMs and these numbers are constant. However, in the real test, the workload fluctuates a bit, for example it takes up to 60 seconds from when a VM is provisioned until it is consuming the maximum amount of resources, because of the boot-up delay. Due to the same reason, it is also illustrated that there is a lag (around 60 seconds) between the curves representing the results in real tests and that of simulations.

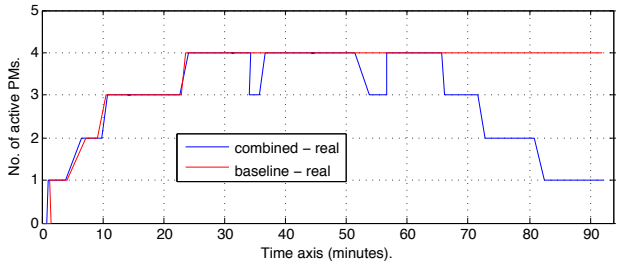

Fig. 14. No. of active PMs (real): combined vs. baseline.

Figure 14 presents the number of active PMs over time during the tests. We can see that the combined algorithm suspends one PM roughly between $35th$ minute and $36th$ minute and one PM roughly between the $54th$ minute and the $57th$ minute, resulting in higher CPU usage of the whole testbed, which is consistent with Figure 12. Additionally, benefiting from VM migration, some PMs can be suspended by the combined algorithm and the CPU usage can stay at a high level even after the $60th$ minute. In contrast, CPU usage achieved by the baseline algorithm keeps dropping as no new VMs arrive after the $60th$ minute and more PMs become idle when VMs terminate.

TABLE V
COMPLEMENTARY RESULTS FOR REAL TESTS: COMBINED VS. BASELINE.

	VMs suspended	Cores suspended	CPU usage (%)
combined	0.001%	<0.0001%	73.58
baseline	1.642%	6.103%	52.96

Table V summarizes some complementary results for the real tests. For the baseline algorithm, an average of 1.642% of VM instances are suspended per minute, while it is only 0.001% for the combined. Looking at the average percentage of VM cores suspended, the difference between the combined and the baseline algorithm is even larger. The reason behind this is that, if it is necessary to suspend VMs, the combined algorithm tends to suspend small instances that with fewer cores. Regarding the average CPU usage during the test, the combined algorithm also outperforms the baseline algorithm.

Finally, we also remark that in the simulations, the total time spent on VM migration was 77 seconds, while the duration in the real-world was 178.2 seconds. This difference is because migration time in the simulations are modeled on post-copy migration which transfers each memory page only once. The real-world tests are run using the standard KVM hypervisor, version 1.5.0, which does not include a post-copy migration algorithm. Because of this, pre-copy migration is used and this type of live migration algorithm uses an iterative transfer where memory pages can be sent multiple times, thus increasing migration time and making it hard to predict the migration time [32].

VI. Related Work

Optimal mapping of admitted VMs to a set of PMs in order to gain maximum profit while complying to all SLAs specified by customers is challenging for cloud providers as it is in general a NP-hard problem [20], [27]. Various algorithms have been proposed to produce near-optimal placement schemes, e.g., by Jing et al. [38], who present an improved genetic algorithm aimed to optimize possibly conflicting objectives, including making efficient usage of multidimensional resources, avoiding hotspots, and reducing power consumption. On the other hand, given the dynamic nature of clouds, with significant changes over time both in workload demands and available resources, the mapping of VMs to PMs need to be revisited regularly. Live migration of running VMs is therefore a necessity. A comprehensive study on principles and performance of live migration mechanisms (including precopy, postcopy and hybrid) is presented in [26], which also discusses how migration downtime can be reduced. Li et al.[21] define a framework for joint optimization of data center deployment, VM assignment, and migration. Based on fluctuations in network performance (latency), they propose a method based on network flow maximization to estimate VM migration cost by amortizing it to the latency of every access. Song et al.[30] define a bin packing approach to allocation and migration of VMs in data centers and similarly, Sato et al.[28] combine bin packing with resource usage prediction to dynamically optimize VM placement. Auto-regressive models are used for predictions and the number of VM migrations is minimized (avoiding ping-pong effects) based on the predictions. Li et al. [19] aim to minimize VM completion time using a knapsack formulation for VM placement. A hybrid on-line off-line scheme is used where VM migrations are combined with the knapsack placement algorithm. A defragmentation approach by Shanmuganathan et al. [29] include two algorithms whereas Avin et al. [3] propose simple destination swap strategies for VMs in order to reduce network traffic.

A large amount of effort has been devoted to server consolidation methods based on workload analysis, aiming at improving efficiency in cloud infrastructures [14], [31], [35]. Additional mechanisms for isolation of resources in hardware [12], [15], or software [34] have been developed to reduce the performance degradation introduced by consolidation of multiple VMs on a same server. Further, Roytman et al. present a polynomial time algorithm to determine the best suited VM combinations to be co-located [27], yielding server energy saving and VM performance preservation. However,these approaches commonly operate in off-line manners which are not able to dynamically and efficiently adapt the cloud to the changes (including workload variations, system failures, etc). They neither take VM pricing schemes and monetary penalty for SLA violation into consideration. Khanna et al. [17] propose a framework to detect application performance deviations and a VM migration mechanism to handle these. They proposed a set of server consolidation heuristics based on VM migration costs and server residual capacity. Xiao et al. [37] introduce some skewness metrics and use these to avoid hot and cold spots in datacenters and migrate VMs around.

In 2001, the Autonomic Computing [11] initiative was initiated by IBM, who also introduced the MAPE-K reference model. Its goal is to build computing systems that can manage themselves given high-level objectives from administrators [16]. In the MAPE-K model, with the support of a Knowledge base, the system Monitors the managed elements, analyzes the data monitored, and finally Plans and Executes suitable actions to ensure the system is in a desired state. Although considerable progress has been achieved in the past few years, the original vision remains unfulfilled as even more complexity is added to the system due to the convergence of new technologies and new applications [8]. The main goal of this work is to maximize the monetary profit of running a datacenter by automatic adaption to both internal and external changes, and thus it is within the scope of autonomic computing. The design and implementation of the proposed system (see Section IV) follows the MAPE-K model.

VII. Conclusions and Future Work

In this paper we present a continuous management approach for cloud infrastructure providers to maximize their PM utilization and increase profits. Based on a set of fundamental management actions, our approach can rearrange the VM to PM mapping during operation to increase the resource utilization of the cloud infrastructure, thereby increase the revenue by prioritizing more profitable workloads and reducing energy consumption. The feasibility and performance of our work, consisting of optimization algorithms and a continuous datacenter consolidation software, is evaluated by simulations and real-world experiments on a testbed. Results indicate that overall utilization of the datacenter is increased using out approach and that power consumption is reduced. The testbed results are consistent with the simulation results, which validates our simulation and indicates that our approach is applicable in real-world scenarios.

We have identified several interesting subjects for the future work, e.g., (i) to investigate the impact of the penalty model on our algorithms, (ii) to extend our work to support other pricing models and to integrate our algorithms in open-source cloud middlewares such as CloudStack and OpenStack, and (iii) to study the feasibility of incorporating our work with auto-scaling and workload prediction techniques, targeting proactive optimization and even better performance.

VIII. Acknowledgments

Financial support has in part been provided by the European Community's Seventh Framework Programme ([FP7/2010-2013]) under grant agreements no. 257115 (OPTIMIS), the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control, and the Swedish Government's strategic effort eSSENCE.

REFERENCES

[1] Sysstat. http://sebastien.godard.pagesperso-orange.fr, visited March 2014.

[2] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS 2012)*, pages 204–212. IEEE, 2012.

[3] C. Avin, O. Dunay, and S. Schmid. Simple Destination-Swap Strategies for Adaptive Intra-and Inter-Tenant VM Migration. *CoRR*, 2013.

[4] L. A. Barroso, C. Jimmy, and U. Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2nd edition, 2013.

[5] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (M'07)*, pages 119–128. IEEE, 2007.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation*, pages 273–286. ACM, 2005.

[7] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation Algorithms for NP-hard Problems. chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[8] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey. Fulfilling the Vision of Autonomic Computing. *Computer*, 43(1):35–41, 2010.

[9] I. Goiri, J. Guitart, and J. Torres. Characterizing Cloud Federation for Enhancing Providers' Profit. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing (CLOUD'10)*, pages 123–130. IEEE, 2010.

[10] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad. Cloud-scale Resource Management: Challenges and Techniques. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, pages 3–3. USENIX Association, 2011.

[11] IBM Corp. An Architectural Blueprint for Autonomic Computing. Technical report, USA, Oct 2004. Tech. Rep.

[12] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. VM^3: Measuring, Modeling and Managing VM Shared Resources. *Computer Networks*, 53(17):2873–2887, 2009.

[13] R. Jones. Virt-top. http://people.redhat.com/~rjones/virt-top/, visited March 2014.

[14] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu. A Cost-sensitive Adaptation Engine for Server Consolidation of Multitier Applications. In *Middleware 2009*, pages 163–183. Springer, 2009.

[15] F. Kamoun. Virtualizing the Datacenter Without Compromising Server Performance. *Ubiquity*, 2009, Aug 2009.

[16] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[17] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application Performance Management in Virtualized Server Environments. In *Proceedings of 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 373–381. IEEE, 2006.

[18] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[19] K. Li, H. Zheng, and J. Wu. Migration-based Virtual Machine Placement in Cloud Systems. In *Proceedings of the IEEE 2nd International Conference on Cloud Networking (CloudNet)*, pages 83–90. IEEE, 2013.

[20] W. Li, J. Tordsson, and E. Elmroth. Virtual Machine Placement for Predictable and Time-Constrained Peak Loads. In *Proceedings of the 8th International Conference on Economics of Grids, Clouds, Systems, and Services (GECON'11)*, 2011. Lecture Notes in Computer Science, Vol. 7150, Springer-Verlag, pp. 120-134.

[21] Y. Li, M. Yao, and C. Lin. Joint Study on Optimizations of Data Center Deployment, VM Assignment and Migration. In *Proceedings of the IEEE/ACM 21st International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2013.

[22] Libvirt. The virtualization API. http://libvirt.org, visited March 2014.

[23] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente. Scheduling Strategies for Optimal Service Deployment across Multiple Clouds. *Future Generation Computer Systems*, 29(6):1431–1441, 2013.

[24] Mark Blackburn, 1E. Five Ways to Reduce Data Center Server Power Consumption, April 2008. White Paper.

[25] L. McVoy. LMbench - Tools for Performance Analysis. http://www.bitmover.com/lmbench/, visited March 2014.

[26] P. Svärd, J. Tordsson, E. Elmroth, S. Walsh, and B. Hudzia. The Noble Art of Live VM Migration - Principles and Performance of Precopy and Postcopy Migration of Demanding Workloads. 2014. Submitted.

[27] A. Roytman, A. Kansal, S. Govindan, J. Liu, and S. Nath. PACMan: Performance Aware Virtual Machine Consolidation. In *Proceedings of the 10th International Conference on Autonomic Computing*, pages 83–94, Berkeley, CA, 2013. USENIX.

[28] K. Sato, M. Samejima, and N. Komoda. Dynamic Optimization of Virtual Machine Placement by Resource Usage Prediction. In *Proceedings of the 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 86–91. IEEE, 2013.

[29] G. Shanmuganathan, A. Gulati, and P. Varman. Defragmenting the Cloud using Demand-based Resource Allocation. In *Proceedings of the ACM SIGMETRICS/international Conference on Measurement and Modeling of Computer Systems*, pages 67–80. ACM, 2013.

[30] W. Song, Z. Xiao, Q. Chen, and H. Luo. Adaptive Resource Provisioning for the Cloud using Online Bin Packing. *IEEE Transactions on Computers*, 99(PrePrints), 2013.

[31] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware Consolidation for Cloud Computing. In *Proceedings of the 2008 Conference on Power aware Computing and Systems*, volume 10. USENIX Association, 2008.

[32] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '11)*, pages 111–120. ACM, 2011.

[33] L. Tomás and J. Tordsson. Improving Cloud Infrastructure Utilization Through Overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC'13, pages 5:1–5:10. ACM, 2013.

[34] A. Verma, P. Ahuja, and A. Neogi. Power-aware Dynamic Placement of HPC Applications. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 175–184. ACM, 2008.

[35] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. Server Workload Analysis for Power Minimization using Consolidation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, pages 28–28. USENIX Association, 2009.

[36] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe. CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines. In *VEE '11: The 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121–132. ACM, 2011.

[37] Z. Xiao, W. Song, and Q. Chen. Dynamic Resource Allocation using Virtual Machines for Cloud Computing Environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117, 2013.

[38] J. Xu and J. A. Fortes. Multi-objective Virtual Machine Placement in Virtualized Data Center Environments. In *Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing*, pages 179–188. IEEE, 2010.