Journal of Engineering and Applied Sciences 14 (24): 9810-9821, 2019
ISSN: 1816-949X

Performance Evaluation of Static VM Consolidation Algorithms for Cloud-based Data Centers with Predefined Machine Types

Young-Chul Shim
Department of Computer Engineering, Hongik University, Seoul, Republic of Korea
shim@cs.hongik.ac.kr

Abstract: Energy efficiency in data centers is a very important issue and getting growing attention from researchers. One approach to reduce energy consumption is to allocate tasks to Virtual Machines (VMs) created in Physical Machines (PMs) in such a way that the number of idle PMs is maximized. Approaches of this kind are called VM consolidation methods. Idle PMs can be put into an energy-saving sleep mode in which PMs consume significantly lower energy than in the normal operation mode. But if too many VMs are packed into a single PM, the performance interference among VMs can cause significant slowdown to jobs. When a new job arrives at a cloud, the tasks of the job should be allocated to idle VMs. If there are enough number of idle VMs, we should decide to which idle VMs those tasks should be assigned. If there are not enough idle VMs, we should create necessary number of idle VMs on proper PMs before allocating the tasks to idle VMs. This problem is called the static VM consolidation problem. In this study, we propose four algorithms for this static VM consolidation problem. When we propose algorithms, we take following issues into considerations: imperfect performance isolation of virtualization technology, flexible and efficient proactive VM creation policy, PMs consisting of multiple CPUs each of which consists of multiple cores and VMs which are created with pre-defined machine types. Further, we assume that we do not have the knowledge of the completion time of a job, although, its resource requirements can be known a priori. We analyze the proposed algorithms through simulation with synthetic workloads obtained by analyzing the characteristics of workloads in real data centers. We measure following three metrics and suggest the best algorithm: ratio of idle PMs, service level agreement violation ratio and the total energy consumption in a cloud.

Key words: Cloud-based data centers, energy-efficiency, inter-VM performance interference, SLA violations, static VM consolidation algorithm, resource requirement

INTRODUCTION

Since, the emergence of the concept of cloud computing, it is getting more widely adopted and deployed in the IT industry sector and receiving more attention from computer scientists and engineers. NIST defines cloud computing to be a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (Mell and Grance, 2011; Singh and Chana, 2016). From a hardware provisioning and pricing point of view, a cloud computing environment has advantages as follow (Armbrust *et al*., 2010).

The appearance of infinite computing resources available on demand, quickly enough to follow load surges, thereby eliminating the need for cloud computing users to plan far ahead for provisioning. The elimination of an up-front commitment by cloud users, thereby allowing companies to start small and increase hardware resources only when there is an increase in their needs.

The ability to pay for use of computing resources on a short-term basis as needed (for example, processors by the hour and storage by the day) and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful. Examples of well-known cloud computing systems include Amazon EC2, Microsoft Azure and Google Compute Engine (Li *et al*., 2010; Lu and Zeng, 2014).

Virtualization is an essential mechanism in providing the computing as a service vision of cloud-based data centers. By providing physical resource sharing, fault isolation, security isolation and live migration, virtualization allows diverse applications to run in isolated environments through creating multiple Virtual Machines (VMs) on shared hardware platforms (Pu *et al*., 2012). When a user rents VMs to run an application on them, he

specifies the amount of resources allocated to each VM by stating required CPU cycles, memory and disk. VM Monitor (VMM) or hypervisor manages and multiplexes access to the physical resources, maintaining isolation between VMs at all times. As the physical resources are virtualized, several VMs, each of which is self-contained with its own operating system can be executed on a Physical Machine (PM) (Mishra *et al*., 2012).

Scheduling in a cloud-based data center is a two stage mapping problem: first on which PMs VMs should be created and then which VMs the tasks of an incoming job should be assigned to. VMs can be created before a job arrives and this strategy is called the proactive VM creation or VM precreation. On the other hand when a new job arrives, VMs needed to run this job are created and this VM management policy is called the reactive VM creation. It is well known that creating a new VM takes several hundred seconds and therefore is a very costly process for short-running jobs (Mann, 2015). When a job is allocated to a set of VMs, each task of that job is assigned to one VM and this VM should have enough resources to run the assigned task. It is also mandatory that the resources needed by all VMs running on a PM should not exceed the capacity of that PM, so that, the PM may not be overloaded. We do not consider over-provisioning in this study.

Another important issue in the scheduling problem is the minimization of the energy consumed in the cloud-based data center. The mapping should be performed in such a way that the total amount of energy consumed by all the PMs which run all the VMs should be minimized. It is reported that worldwide, the data centers use about 30 billion kilowatts of electricity, roughly equivalent to the output of 30 nuclear power plants (Glanz, 2012). And this number is expected to grow 12% a year. This excessive use of energy in data centers not only raises the operation cost of data centers heavily but also creates environmental issues such as air pollution. Therefore, minimizing energy consumption in data centers is becoming a more and more important issue. In many literatures, the power consumption of a server is modeled in its simplest form as follow:

$$P(u) = P_{min} + (P_{max} - P_{min}) \cdot u$$

Where:

- u = The CPU utilization
- P_{min} = The Power consumed when the server is idle
- P_{max} = The Power consumed when the server is fully utilized

In this study we use two terminologies, PMs and servers, interchangeably. For most of all the servers available in the current market, it is well-known that P_{min} is almost 50% or more of P_{max} (Barroso and Holzle, 2007). We can think of turning off a server when it becomes idle. But the procedure of turning off and then on a server takes too much time and therefore cannot be thought of as a practical solution. But some researchers developed a power-saving mechanism in which an idle server can be rapidly put into a sleep mode and consume just 10% of P_{min}. When a job arrives at a server, it can be put back into the normal operation mode rapidly (Meisner *et al*., 2009). With this kind of power saving mechanism, a server consumes power with the above power model when it has one or more tasks to run but can consume very little power when it has no tasks.

With the above observation, if we run all the VMs on the smallest number of PMs and therefore, maximize the number of idle PMs, we can greatly reduce the amount of energy consumed in data centers. This feature of scheduling, packing VMs into a small number of PMs is called VM consolidation. It is known that VM consolidations are classified into:

- Static consolidation
- Semi-static consolidation
- Dynamic consolidation (Verma *et al*., 2014)

In static consolidation the sizing and placement of VMs on PMs is determined statically when a job arrives and does not change over a period of time. Semi-static consolidation attempts to take advantage of medium to long term workload variations by periodically re-sizing workloads and relocating them on target PMs. Semi-static consolidation typically takes advantage of intra-week variations or intra-month variations. Semi-static consolidation is performed by re-sizing and relocating workloads once a week or once a month. Dynamic VM consolidation extends the idea of semi-static consolidation even further by consolidating workloads daily or multiple times on the same day. Consolidations on such a frequent basis cannot be performed using VM relocation due to downtime required for VM relocation and therefore, need live VM re-sizing and live VM migration. While semi-static consolidation is suitable for very long jobs running for several weeks or months and dynamic consolidation is suitable for long jobs running for several days, static consolidation will be suitable for relatively short jobs running for a couple of hours or days.

In this study, we consider an energy-efficient job scheduling problem in data centers as a static consolidation problem. The static consolidation problem

has been studied extensively by many researchers as will be described in the next section on related works. But most of their works have the following problems.

They assume that the performance of a VM is not affected by other VMs running on the same PM. But recent research results which will be summarized in the next section, show that current virtualization technology does not provide perfect performance isolation. This imperfect performance isolation will have the effect of making job completion time longer than calculated assuming perfect performance isolation and in some cases can cause Service Level Agreement (SLA) violations for some jobs.

Many researchers assume that when a job arrives, it will find VMs whose resource capacities are just right for the job if the data center has enough resources for the job. In case of the reactive VM creation policy, the job should wait several hundred seconds before the required VMs are created and therefore this assumption is not valid. But with the proactive VM creation policy, this assumption can be valid as long as the data center has enough resources to run the submitted job. But the decision of how many VMs should be created and maintained proactively and where they should be located must be handled very carefully because the over-creation of VMs will reduce the number of idle PMs and therefore, decrease the energy-efficiency of the whole data center while under-creation of VMs may cause the delay when starting the job execution.

Many researchers assume that a server consists of just one very powerful core to make the problem simple. But in reality a server can have multiple CPUs and each CPU has multiple cores. This makes the problem of analyzing the performance interference among VMs running on a server very complicated because the performance of a VM will be affected by three different types of VMs on the same server: VMs running on the same core, VMs running on the different cores of the same CPU and VMs running on the different CPUs in the same server. This results from the difference in the amount of shared hardware resources among VMs in the above three cases.

Many energy-efficient static consolidation algorithms assume that we have the knowledge of both the completion time of a job and its resource requirements. But the job completion time varies significantly depending upon the amount of inputs and is hard to predict while approximate resource requirements of a job can be predicted relatively easily. In this study we present and analyze several static VM consolidation algorithms considering:

- Imperfect performance isolation of virtualization technology
- Flexible and efficient proactive VM creation policy
- Servers consisting of multiple CPUs, each of which consists of multiple cores
- VMs which are created with pre-defined machine types (Anonymous, 2019a-c)

Although, we assume that we have approximate knowledge of resource requirements of jobs, we do not require that we know the completion time of jobs a priori. Modeling the static VM consolidation problem as a bin packing problem as will be explained in the next section, we introduce 4 static VM consolidation algorithms:

- Best fit
- Upper-bounded best fit
- Upper-bounded load-balancing
- Upper-bounded and lower-bounded load-balancing

When a new job arrives, the data center first tries to find and assign a set of idle VMs for the job from the idle VM pool of the proper machine type. If the data center cannot find such a set, it initiates the creation of necessary idle VMs. During this process the above algorithms are applied to:

- Select idle VMs for the newly submitted job
- Decide on which PM a new idle VMs should be created if necessary

In the simulation-based analysis of the above 4 algorithms, we use a proactive VM creation policy which is efficient in reducing both job execution time and energy usage in the whole data center. The proposed VM creation policy tries to reduce the possibility that a newly arriving job may not find a proper set of idle VMs and at the same to maintain the number of idle VMs as low as possible in order to minimize energy consumption required to maintain idle VMs. Through simulation using realistic server hardware modeling and synthetic workload based upon statistics derived from real data center workloads, we measure values of three metrics for each algorithm: the idle PM ratio, the SLA violation ratio and the total amount of energy consumed in the data center while varying the workload level on the data center.

Literature review: The VM consolidation problem is often formulated as a bin packing problem which can be described as follows. Given n items and m bins with:

w_j = weight of item j

c = capacity of each bin

Assign each item to one bin, so that, the total weight of the items in each bin does not exceed c and the number of bins used is minimum (Martello and Toth, 1990). This problem is known to be NP-hard and several approximate algorithms are introduced:

- First-Fit (FF)
- Best-Fit (BF)
- First-Fit Decreasing (FFD)
- Best-Fit Decreasing (BFD) algorithms

We assume that items and bins are indexed. The FF algorithm considers the items according to increasing indices and assigns each item to the lowest indexed bin which it fits. The BF algorithm is obtained from FF by assigning the current item to the feasible bin having the smallest residual capacity. If items are sorted in the non-increasing order of their weight and then FF and BF algorithms are applied, the resulting algorithms are called FFD and BFD algorithms, respectively. Making an item, w_j, a bin and c correspond to a VM, the resource requirements of VM_j, a PM and the resource capacity of a PM, respectively, the static VM consolidation problem exactly becomes a bin packing problem. When a job, each of which requires several VMs of the same resource requirements, arrives one at a time, either FF or BF algorithms are used. If many jobs should be scheduled simultaneously and resource requirements of tasks of a job may be different from resource requirements of tasks of other tasks, FF, BF, FFD or BFD algorithms can be applied. In this study we address the situation where jobs arrive one at a time and are scheduled independently from other jobs and therefore, we do not need to consider using FFD or BFD.

Eucalyptus which is open source software for building AWS-compatible clouds, provides the following VM allocation algorithm: a greedy algorithm, a round-robin algorithm and a power save algorithm. The greedy algorithm is an FF algorithm explained above. The round robin algorithm mainly focuses distributing the load equally to all the PMs. The power save algorithm optimizes the power consumption by turning off PMs which are not currently used which is considered impractical.

Lin *et al.* (2011) proposed a hybrid approach which combines FF and dynamic round robin algorithms. The dynamic round-robin algorithm uses two rules to help consolidate VMs. The first rule is that if a VM has finished and there are still other VMs hosted on the same PM, this PM will accept no more new VMs. Such PMs are referred to as being in the retiring state. The second rule is that if a PM is in the retiring state for a sufficiently long period of time, instead of waiting for the residing VMs to finish, the PM will be forced to migrate the rest of the VMs to other PMs and be shut down after the migration is finished. The hybrid approach uses the FF algorithm during rush hours and dynamic round-robin during non-rush hours. This approach is a combination of static and dynamic VM consolidation methods.

Lee and Zomaya (2012) describes two energy saving static VM consolidation algorithms: ECRC and MaxUtil. When a new job which is to be run on a new VM arrives, a cost function is computed for each PM and the PM which has the lowest cost is selected to run the VM. Beloglazov *et al.* (2012) introduces a modified BFD algorithm which allocates each VM to a PM that provides the least increase of power consumption due to this allocation. The algorithm is used to allocate multiple VMs for multiple jobs simultaneously. The algorithms by Lee and Zomaya (2012) and Beloglazov *et al.* (2012) assume that the exact completion time of a job is known a priori and is not affected by other VMs collocated on the same PM.

Although, none of the works that, we described above consider the performance interference among VMs running on the same PM, many researchers have studied this performance interference issues. Koh *et al.* (2007), Govindan *et al.* (2011) and Oh *et al.* (2011) studied this issue by measuring how much performance degradation a VM would cause to other VM being executed on the same PM. But all of them focus on performance impact between only two VMs in a PM and their results do not generalize to the case of many VMs running simultaneously on one PM.

Lin *et al.* (2016) propose an energy-efficient task scheduling algorithm in a data center. They introduce a mathematical model for inter-VM performance interference which can be used even when more than two VMs are running in a PM. But their scheduling algorithm is based on the assumption that a priori execution time of a task is known.

There are works on characterizing workloads on cloud systems (Mishra *et al.*, 2010; Reiss *et al.*, 2012]. We utilize these results to generate cloud workloads for our simulation-based analysis.

MATERIALS AND METHODS

Proposed algorithms for static VM consolidation: In this study, we describe characteristics for data center workloads, a VM pre-creation policy and 4 algorithms for static VM consolidation.

The data center workload consists of jobs submitted to the data center. A job is characterized by the triple

(duration, resource requirements of a task, number of tasks). The duration represents how long the job will run without interference from other jobs. From the Google computer cluster workload trace analysis, Mishra *et al.* (2010) finds that job durations are bimodal, meaning that somewhat <30 min or larger than 18 h. Mishra *et al.* (2010) also observes that a majority of jobs last for only minutes while some jobs run for very long time from the similar trace analysis. We classify the jobs into two classes according to the duration: short and long. The resource requirements of a task show how much vCPU (virtual CPUs) and memory a task requires and by this amount tasks are classified into three classes: small, medium and large. The number of tasks means how many tasks a job consists of. Trace analyses show that the number of jobs decreases exponentially as the number of constituent tasks increases (Reiss, 2012). The total amount of resources for a job is calculated by multiplying the resource requirements of a task by the number of tasks. Mishra *et al.* (2010) shows that jobs shorter than two hours account for <10% of the overall resource utilization even though they represent more than 95% of the job counts. From these observations we infer that a very large number of short-running jobs require a small amount of resources and consist of a few tasks while a small number of long-running jobs require medium to large amount of resources and consist of medium to large number of tasks.

In most cases VMs are created with a predefined machine type which specifies a particular collection of virtualized hardware resources available to a VM (Anonymous, 2019a-c). The specification is made in terms of the number of vCPUs and the memory size. Like the task classification explained above, we define three machine types for VMs:

- Small
- Medium
- Large

Each task will be executed on a distinct VM with a matching machine type. So when a job arrives, it should receive a proper collection of idle VMs before it can start. A proper collection of idle VMs means that there should be enough number of VMs of the matching machine type. Otherwise the job should wait until such a proper collection becomes available. It is well known that the creation of a new VM is time consuming process taking several hundred seconds. This VM creation time will cause serious performance degradation especially to the short jobs whose number dominates in the data center workload. To avoid this undesirable situation, it is necessary to maintain a sufficient number of idle VMs for each machine type or equally for each task type. From the statistics of the data center workloads we can predict how many idle VMs will be needed for each machine type, to execute the jobs that will arrive during the average VM creation time. The VM precreation policy tries to maintain this number of idle VMs for each machine type. Our VM precreation policy is based upon this statistics-based prediction and tries to maintain this number of idle VMs for each machine type. When a new job is allocated to VMs of a proper type, the number of idle VMs of this type decreases and therefore additional idle VMs are created as necessary. When a job is completed, the number of idle VMs of the machine type proper for the job increases and therefore excessive VMs are deleted. With the above mentioned VM precreation policy, the skeleton of the overall scheduler looks as follow:

```
scheduler(eventType) {
  if (eventType is jobArrival) {
    /* we assume that tasks of the new job is assigned to the smallest VMs
whose resource capacity */
    /* is not smaller than the resource requirement of the task */
    if (there are enough number of idle VMs for tasks of this new job)
      assign each task of the job to an idle VM;
    else /* there are not enough number of idle VMs */
      create necessary number of VMs for the tasks of this new job;
        /* tasks will be assigned to VMs when all the requested VMs are
created */
    calculate the number of idle VMs to satisfy the VM precreation policy
and
    create idle VMs to satisfy that number;
  } else /* eventType is jobTermination */ {
    return the VMs, that were assigned to the terminated job, to the idle
VM pool;
    calculate the number of idle VMs to satisfy the VM precreation policy
and
    destroy idle VMs exceeding that number;
  }
}
```

The above scheduler requires four functions as follow:

Assign tasks to VMs: There are enough number of VMs for tasks of a new job. Select a proper set of VMs required to run the tasks of this new job.

Create VMs for tasks: There has been an unexpected surge of creation of jobs requiring VMs of the same type, so there are not enough number of VMs for a new job. It is required to create the deficient number of VMs. If there are N tasks and M idle VMs s.t N>M, first select M tasks and assign them to the M idle VMs. Then create N-M VMs and assign remaining N-M tasks to them.

Create Idle VM: Some idle VMs were assigned to a new job, so, it is necessary to create new idle VMs to satisfy the VM precreation policy. If N new idle VMs are required, this function is called N times.

Destroy idle VM: A job was terminated and the VMs assigned to that job were returned to the idle VM pool and therefore, there may be more idle VMs than the VM precreation policy demands. These overflow VMs are terminated. If N idle VMs are to be destroyed, this function is called N times. For the implementation of the above four functions, in this study we consider the following four policies:

- Best Fit (BF)
- Upper-Bounded Best Fit (UBBF)
- Upper-Bounded Load-Balancing (UBLB)
- Upper-Bounded and Lower-Bounded Load Balancing (UBLBLB)

Now, we explain the above policies. In the discussion we use the following terminologies. If a PM have M VMs then it can execute N tasks such that $M \geq N$. The capacity of a VM is the maximum workload that it can accommodate. The sum of the capacity of M VMs in a PM is called the possible workload of the PM and the sum of workload of N tasks in that PM is the actual workload of the PM. We assume that a new job to be scheduled consists of N tasks and each task has workload p. A task is assigned to an idle VM of the machine type that has the smallest capacity λ such that $\rho \leq \lambda$.

The purpose of the best fit policy is to maximize the number of idle PMs in the cloud. To achieve its goal it assigns a task to the idle VM in the PM with the highest actual workload, creates a new VM in the PM with the highest actual workload and deletes VM from the PM with the lowest actual workload. The algorithms for these functions are as follow:

```
assignTasksToVMs (Task set, pool of idle VMs of proper machine type)
{
  /* if there are N tasks and M idle VMs in the cloud such that N≤M, this
function is called, */
  /* and task assignment is completed */
  while (Task set is not empty) {
    retrieve one task from the task set
    sort idle VMs in such a way that the actual workload of PM running the
VM is in decreasing order
    assign the task to the first VM in the pool and remove the VM from the
pool
    increase the actual workload of the selected PM by the load of the task
ρ
  }
}
createVMsForTasks (Task set) {
  /* if there are N tasks and M idle VMs in the cloud such that N>M, this
function is called */
  select M tasks from the Task set and assign them to M idle VMs using
the above function;
  put remaining N-M tasks into the unassigned task set
  /* Unassigned task set has N-M tasks */
  While (Unassigned task set is not empty) {
    retrieve one task from the unassigned task set
    select only those non-idle PMs whose possible workload can accept the
workload λ
      and sort them in the decreasing order of actual workload
    if (there is no PM in the PM list)
      select any idle PM and put it into the PM list
    retrieve the first PM, create a new VM in that PM and assign the task to
that VM
    increase the possible workload of the selected PM by λ
    increase the actual workload of the selected PM by ρ
  }
createIdleVM (VM type) {
  /* create a VM with the capacity λ */
  select only those non-idle PMs whose possible workload can accept the
new workload λ
      put them into the selected PM list
      and sort them in the decreasing order of actual workload
  if (selected PM list is not empty)
    select the PM with the highest actual workload
  else
    select any idle PM
  create a new VM in the selected PM
  increase the possible workload of the selected PM by λ
}
destroyIdleVM (VM type) {
  select only those PMs which has an idle VM of the specified type
  sort PMs in the increasing order of actual workload
  destroy the idle VM of the specified type from the selected PM
  decrease the possible workload of the selected PM by λ
  if (the possible workload of the selected PM becomes 0)
    make the PM an idle PM
}
```

The best fit policy tries to put more work on the PM with the highest actual workload. The problem with this policy is that as the workload of a PM increases, the performance interference among tasks running in that PM increases and therefore, the completion time of those tasks will be delayed. If the workload becomes too much high, the tasks in that PM can experience excessive delay resulting in the SLA violation for the jobs which have these tasks as their component tasks. To avoid this undesirable situation, the upper-bounded best fit policy sets the upper bound on the actual workload on PMs, so that, the actual workload of a PM cannot exceed this upper bound. With this policy although there are N tasks and M idle machines such that $N \leq M$, we do not know whether tasks can be assigned to idle VMs without violating the upper bound on the actual workload. Therefore, when tasks are assigned to VMs, regardless of whether there are enough number of idle VMs or not, the assign tasks to VMs function which is modified as follow is applied first:

```
assignTasksToVMs (Task set, pool of idle VMs of proper type) {
  /* a PM has an upper bound UB on its actual workload */
  remove from the idle VM pool those VMs whose PM has the actual
workload > UB-ρ
  while ((Tasks set is not empty) and (idle VM pool is not empty)) {
    retrieve a task from the task set
    sort idle VMs in such a way that the actual workload of PM running the
VM is in decreasing order
    assign the task to the first VM in the VM pool and remove that VM
from the pool
    increase the actual workload of the selected PM by ρ
    if the actual workload of that PM exceeds UB-ρ, remove all the VMs
```

```
running in that PM from
        the idle VM pool
  }
  if (Task set is empty)
    task allocation is completed
  else
     the task set becomes the unassigned task set and the following function
is called
 }
 createVMsForTasks (Unassigned task set) {
   /* L tasks have been assigned to idle VMs using the above algorithm. */
   /* The unassigned task set has N-L tasks */
    select only those non-idle PMs whose possible workload can accept the
workload λ
      and has actual workload≤UB-ρ
   put those PMs in the PM pool
   while (Unassigned task set is not empty) {
     if (PM pool is empty)
        select any idle PM and put it into PM pool
     sort PMs in the decreasing order of actual workload
     select PM with the highest actual workload, create a new VM in that PM
and assign the task to that VM
     increase the possible workload of the selected PM by λ
     increase the actual workload of the selected PM by ρ
     if ((the actual workload of the selected PM >UB-ρ) or
       (the selected PM cannot accept any more VM with the capacity λ))
        remove that PM from the PM pool
   }
 }
```

Two functions, createIdleVM and destoryIdleVM are the same as in the case of the best fit policy. The third policy, upper-bounded load-balancing, tries to distribute workload on as many PMs as possible to minimize the performance interference among tasks running in the same PM. At the same time this policy avoids the situation such that a PM's actual workload exceeds an upper bound as in the upper-bounded best fit policy. Although, it may be able to reduce the job completion time, it may reduce the number of idle PMs seriously. To avoid this undesirable situation, we try to use idle PMs in only unavoidable cases. An idle PM is used when none of the non-idle PMs can accept a task or an idle VM. As in the case of the upper-bounded best fit policy, although, there are N tasks and M idle machines such that N≤M, we do not know whether tasks can be assigned to idle VMs without violating the upper bound on the actual workload. Therefore when tasks are assigned to VMs, regardless of whether there are enough number of idle VMs or not, the assignTasksToVMs function which is modified as follow is applied first.

```
 assignTasksToVMs(Task set, pool of idle VMs of proper type) {
   /* a PM has an upper bound UB on its actual workload */
     remove from the idle VM pool those VMs whose PM has the actual
workload >UB-ρ
   while ((Task set is not empty) and (idle VM pool is not empty)) {
     retrieve a task from the task set
     sort idle VMs in such a way that the actual workload of PM running the
VM is in increasing order
      assign the task to the first VM in the pool and remove the VM from the
pool
     increase the actual workload of the selected PM by ρ
     if (the actual workload of that PM exceeds UB-ρ)
        remove all the VMs running in that PM from the idle VM pool
   }
   if (Task set is empty)
     task allocation is completed and
   else
      the task set becomes unassigned task set and the following function is
called
 }
 createVMsForTasks(Unassigned task set) {
   /* L tasks have been assigned to idle VMs using the above algorithm. */
   /* the unassigned task set has N-L tasks */
    select only those non-idle PMs whose possible workload can accept the
workload λ
      and has actual workload ≤UB-ρ
   put those PMs in the PM pool
   while (Unassigned task set is not empty) {
     retrieve a task from the Unassigned task set
     if (PM pool is empty)
        select any idle PM and put it into PM pool
     sort PMs in the increasing order of actual workload
     select PM with lowest actual workload, create a new VM in that PM and
assign the task to that VM
     increase the possible workload of the selected PM by λ
     increase the actual workload of the selected PM by ρ
     if ((the actual workload of the selected PM >UB-ρ) or
       (the selected PM cannot accept any more VM with the capacity λ))
        remove that PM from the PM pool
   }
 }
 createIdleVM (VM type) {
   /* creates a VM with the capacity λ */
     select only non-idle PMs whose possible workload can accept the new
workload λ
       and sort them in the increasing order of actual workload
   if (selected PM list is not empty)
     select the PM with the lowest actual workload
   else
     select any idle PM
   create a new VM in the selected PM
   increase the possible workload of the selected PM by λ
 }
 destroyIdleVM (VM type) {
   select only those PMs which has idle VMs of the specified type
   sort PMs in the increasing order of actual workload
    destroy the idle VM of the specified type from the PM with the lowest
actual workload
   decrease the possible workload of the selected PM by λ
   if (the possible workload of the selected PM becomes 0)
     make the PM an idle PM
 }
```

As an effort to increase an idle PMs in the upper-bounded load-balancing policy, the upper-bounded and lower-bounded load balancing policy sets a lower bound, LB, on the actual workload on PMs, so that, new tasks are assigned or new idle VMs are created on PMs whose actual workload is not lower than the lower bound if possible. Those PMs with actual workload lower than the lower bound will gradually lose active tasks and idle VMs and finally become an idle PM. With this policy the functions in the upper-bounded load balancing policy is modified as follow:

```
assignTasksToVMs(Task set, the pool of idle VMs of proper type) {
  /* a PM has an upper bound UB and a lower bound LB on its actual workload */
  remove from the idle VM pool those VMs whose PM has the actual workload < LB or
    has the actual workload >UB-ρ
  while ((Tasks set is not empty) and (idle VM pool is not empty)) {
    retrieve a task from the task set
    sort idle VMs in such a way that the actual workload of PMs running the VM is in increasing order
      assign the task to the first VM in the VM pool and remove that VM from the pool
    increase the actual workload of the selected PM by ρ
    if (the actual workload of that PM exceeds UB-ρ)
      remove all the VMs running in that PM from the idle VM pool
  }
  if (Task set is empty)
    task allocation is completed and return
  /* Now try to allocate tasks to PMs whose actual workload is lower than LB */
  from the non-idle PMs whose actual workload is lower than LB collect all the proper VMs
    and put them into idle VM pool
  while ((Task set is not empty) and (idle VM pool is not empty))
    retrieve a task from the task set
    sort idle VMs in such a way that the actual workload of PMs running the VM is in decreasing order
      assign the task to the first VM in the VM pool and remove that VM from the pool
      /* among the PMs with actual workload < LB, a VM in the PM with the highest actual workload *?
    /* is selected */
    increase the actual workload of the selected PM by ρ
    if (the actual workload of that PM exceeds UB-ρ)
      remove all the VMs running in that PM from the idle VM pool
  }
  if (Task set is empty)
    task allocation is completed and return
  else
    task set becomes Unassigned task set and the following function is called
}
createVMsForTasks(Unassigned task set) {
  /* L tasks have been assigned to idle VMs using the above algorithm */
  /* the unassigned task set has N-L tasks */
  collect only those non-idle PMs whose possible workload can accept the workload λ
    and has actual workload≤UBρ and >LB
  put those PMs in the PM pool
  while ((Unassigned task set is not empty) and (PM pool is not empty)) {
    retrieve a task from the Unassigned task set
    sort PMs in the increasing order of actual workload
    select PM with lowest actual workload, create a new VM in that PM and assign the task to that VM
    increase the possible workload of the selected PM by λ
    increase the actual workload of the selected PM by ρ
    if ((the actual workload of the selected PM>UB-ρ) or
      (the selected PM cannot accept any more VM with the capacity λ))
      remove that PM from the PM pool
  }
  if (Unassigned task set is empty)
    task allocation is completed and return
  /* Now try to allocate tasks to PMs whose actual workload is lower than LB */
  collect non-idle PMs whose possible workload can accept the workload λ
    and has actual workload < LB
  put those PMs in the PM pool
  while (Unassigned task set is not empty) {
    if (PM pool is empty)
      select any idle PM and put it into PM pool
    sort PMs in the decreasing order of actual workload
    select the first PM, create a new VM in that PM and assign the task to that VM
    increase the possible workload of the selected PM by λ
    increase the actual workload of the selected PM by ρ
    if ((the actual workload of the selected PM>UB-ρ) or
      (the selected PM cannot accept any more VM with the capacity λ))
      remove that PM from the PM pool
  }
}
createIdleVM (VM type) {
  /* create an idle VM with the capacity λ*/
  collect non-idle PMs which can accept a new idle VM with capacity λ and
    whose actual capacity lies between LB and UB-λ
  if (found) {
    select the PM with the lowest actual workload
    create a new idle VM on that PM
    increase the possible workload of the PM by λ
    return
  }
  collect non-idle PMs which can accept a new idle VM with capacity λ and
    whose actual capacity is lower than LB
  if (found) {
    select the PM with the highest actual workload
    create a new idle VM on that PM
    increase the possible workload of the PM by λ
    return
  }
  select any idle PM
  create a new idle VM on the selected idle PM
  increase the possible workload of the PM by λ
}
destoryIdleVM (VM type) {
  collect non-idle PMs whose actual workload is lower than LB and have the idle VM of
    the given machine type
  if (found) {
    select the PM with the lowest actual workload
    destroy idle VM from that PM
    decrease the possible workload of that PM by λ
    if (the possible workload of the selected PM becomes 0)
      make the PM an idle PM
    return
  }
  collect non-idle PMs which have the idle VM of the given machine type
  if (found) {
    select the PM with the lowest actual workload
    destroy idle VM from that PM
    decrease the possible workload of that PM by λ
    if (the possible workload of the selected PM becomes 0)
      make the PM an idle PM
  }
}
```

RESULTS AND DISCUSSION

Simulation results: In this study, we first explain simulation environments for analyzing the performance of proposed algorithms. Then we present the experimental results for the 4 algorithms explained in the previous study.

For simulation we consider a data center consisting of 64 homogeneous Pms. Each PM has 2 CPUs, each CPU has 16 cores and each core is enabled with hyper-threading. VMs are created in one of three predefined machine types:

- Small
- Medium
- Large

A small VM has one vCPU(virtual CPU), a medium VM has four vCPUs and a large VM has sixteen vCPUs. A virtual CPU is implemented as a single hardware hyper-thread. Jobs are characterized by their resource requirements and duration and they are classified into 6 categories:

- Short-small
- Short-medium
- Short-large
- Long-small
- Long-medium
- Long-large

A small job consists of small number of small tasks, a medium job consists of medium number of small tasks or small number of medium tasks and a large job consists of medium number of medium tasks or small number of large tasks. A small task is assigned to a small VM, a medium task is assigned to a medium VM and a large task is assigned to a large VM. A job inter-arrival time is determined in such a way that small jobs comprise 80% of the total job numbers while they consume 10% of total resources, medium jobs comprise 10% of the job numbers while they consume 10% of total resources and large jobs comprise 10% of the job number while they consume 80% of total resources.

If a task is sharing a PM with other tasks, its performance will be degraded, meaning that its execution time will get longer, due to the interference from other tasks. To formulate this performance interference, we borrow equation for performance interference by Lin *et al.* (2016) and slightly modify it. Let T be the execution time of a task without any performance interference. If this task is executed with other tasks in the same PM and the workload due to other tasks is u, then the execution time of the task with performance interference, T', is formulated as follow:

$$T' = T/\left(1 - u^{1/\beta}\right)$$

where, β is the high-load penalty factor and β (0, 1). Note that the load u does not include the load of the task whose execution time is calculated. If u includes the workload of all the tasks in a PM, then the equation will derive an erroneous result that even though there are no other interfering tasks, a task's performance is interfered by itself.

To calculate the energy consumed in a PM, we use equation for the energy consumption model stated in section 1. We let $P_{min} = 0.5 \times P_{max}$ and the energy consumption model becomes as follows:

$$P(u) = 0.5 \times (u+1) \times P_{max}$$

where, u is the actual workload of a PM. If there is at least one task and/or idle VM in a PM, the energy consumption in the PM is defined as above. When a PM is idle, meaning that it does not have neither a task nor an idle VM, we assume that it is put into a sleep mode quickly and consumes as low as 10% of P_{min} or 5% of P_{max}. When a VM is created on this PM, the PM can quickly return to the normal operation mode as suggested by Mesiner.

For the simulation we vary the load level of a data center. The load level is calculated as follow. For each workload class, we calculate the workload due to the workload class as follow:

$$(\text{Workload due to a job}) \times (\text{Average No. of jobs per sec})$$

where the average number of jobs per sec correspond to the inverse of the inter-job arrival time. Then, we sum up all the workloads for 6 workload classes and then divide this number by the number of PMs. In this study, we consider four load levels: 0.25, 0.375, 0.5 and 0.625. The load level 0.25 means that the data center is lightly loaded while 0.625 means that the data center is heavily loaded. From the simulation we measure three metrics:

- The ratio of idle PMs
- The ratio of SLA violations
- The total amount of energy consumed in the data center during the simulation

The ratio of idle PMs is obtained as follows. First the total idle time of each PM during the simulation is calculated and this per-PM idle time is summed up for all the PMs. Then this sum is divided by the number (simulation time)×(total No. of PMs). The ratio of idle PMs is a rough indicator of the energy efficiency of the simulated algorithm and takes the value from 0-1. The larger this ratio gets, the more idle PMs there are in the data center during the simulation and therefore, less energy consumption.

The actual completion time of a job during the simulation is defined to be the time between the arrival time of the job and the finish time of the job. If a job is allocated to VMs as soon as it arrives, the actual completion time of the job consists of only its actual

Table 1: The ratio of idle PMs (%)

Parameters	0.25	0.375	0.5	0.675
BF	62.5	45.4	27.2	10.1
UBBF	59.3	43.7	25.1	8.8
UBLB	55.0	39.2	21.9	5.2
UBLBLB	58.0	42.1	24.7	8.6

Table 2: The ratio for violation (%)

Parameters	0.25	0.375	0.5	0.675
BF	5.4	9.5	15.4	22.3
UBBF	3.2	4.9	7.2	10.4
UBLB	2.9	3.7	4.6	5.2
UBLBLB	3.0	4.1	5.1	5.7

Table 3: Total energy consumption

Parameters	0.25	0.375	0.5	0.675
BF	25984	34841	45238	54272
UBBF	26273	35024	45714	53941
UBLB	26984	36137	46382	54726
UBLBLB	26041	34905	45103	53176

execution time. But if a job has to wait in the job queue for a proper collection of idle VMs, the actual completion time consists of the waiting time and the actual execution time. As already explained earlier, the execution time can become longer than its ideal execution time due to the performance interference from other tasks running in the same PMs. The slowdown of a job is defined to be the ratio of the actual completion time to the ideal execution time. The slowdown has a value ≥1. If the slowdown value gets higher than some threshold, we say that the SLA of the job is violated. In this study, we choose 1.25 for the threshold. To calculate the ratio of SLA violation, we count the number of completed jobs whose SLA has been violated and then divide this number by the total number of completed jobs during the simulation. The SLA violation ratio gets value from 0 to 1. The smaller this ratio gets, the more reliable the consolidation algorithm becomes.

Each run of the simulation is terminated when 50,000 jobs are completed. Table 1-3 show the simulation results. In tables, columns represent load levels and rows represent the consolidation algorithms as follows:

- BF (Best Fit)
- UBBF (Upper-Bounded Best Fit)
- UBLB (Upper-Bounded Load Balancing)
- UBLBLB (Upper-Bounded and Lower-Bounded Load Balancing)

Table 1 shows the idle PM ratio (%) of the proposed VM consolidation algorithms under varying load levels. BF algorithm has the highest idle PM ratio, UBBF and UBLBLB algorithms have almost similar results which are slightly worse than the BF algorithm and UBLB has the worst result. BF and UBBF algorithms try to allocate a new task and/or new idle VM to already packed PM and try to maximize the number of idle PMs. UBBF algorithm has slightly lower idle PM ratio than BF algorithm because UBBF algorithm does not allocate a new task and/or new idle VM on PMs whose actual workload exceeds the upper bound on the actual workload to avoid the situation where tasks which are allocated to a PM with excessively high workload, suffer from too much performance interference and therefore, experience too much delay in their completion time. UBLB and UBLBLB algorithms try to allocate a new task and/or new idle VM to lightly loaded non-idle PM. But they do not allocate a new task or idle VM to highly-loaded PMs. Moreover, they try to utilize non-idle PM as much as possible and therefore, their ratios of idle PMs do not drop drastically. We see the UBLB algorithm has the worst result but UBLBLB algorithm has results very similar to the UBBF algorithm. This is because the UBLBLB algorithm tries to avoid allocate a new task and/or idle VM on PMs with the actual workload lower than lower bound and also tries to destroy idle VMs on the non-idle PMs with the actual workload lower than the lower bound.

Table 2 presents the SLA violation ratio (%) of 4 consolidation algorithms under varying load levels. We see that the SLA violation ratio increases as the load level increases in all the algorithms. Table 2 shows that BF algorithm exhibits the worst violation ratio. This result is quite obvious because the BF algorithm tries to allocate a new task and/or idle VM to already packed PMs and tasks allocated on these packed PMs suffer from serious performance interference. The UBBF algorithm improves the SLA violation ratio drastically because it avoids to allocate a new task and/or idle VM to too highly loaded PMs. The UBLB and UBLBLB algorithms show much lower SLA violation ratio than UBBF because new tasks and/or idle VMs tend to be allocated to non-idle PMs with lower actual workloads. UBLB and UBLBLB algorithms achieve the goal of evenly distributing workloads among non-idle PMs and therefore, the workload of these non-idle PMs increases almost at the same rate and the chances that a certain PM is highly loaded are delayed as much as possible. The UBLBLB algorithm has slightly higher violation ratio than the UBLB algorithm because it tends to use less PMs.

Table 3 shows the total energy consumption of the algorithms under varying load levels. The unit of the consumed energy is P_{max} as explained before. From Table 1 and 3, we see that as the idle PM ratio rises, the consumed energy falls. Under low workload the BF algorithm consumes least energy because it uses least number of PMs but under very high workload it consumes more energy than UBBF and UBLBLB algorithms because those tasks allocated on highly loaded PMs suffer from serious performance interference and their completion time is delayed seriously. Therefore, these highly loaded PMs are utilized for longer time and the consumed energy

in these PMs increases. The UBLB algorithm consumes more energy than other algorithms under any level of workload because this algorithm uses much more PMs than other algorithms. The UBBF and UBLBLB algorithms consume slightly higher amount of energy than the BF algorithm under low workload. But they consume similar or lower amount of energy than the BF algorithm under high workload level. It is because although, they utilize slightly more PMs than the BF algorithm but the chances that tasks are delayed excessively is much lower than in the BF algorithm. Between UBBF and UBLBLB algorithms the UBLBLB algorithm consumes less energy. It is because both algorithms use similar number of PMs but the chances that some tasks are seriously delayed due to performance interference is much lower in the UBLBLB algorithm than the UBBF algorithm.

We want to minimize the energy consumption in a data center but at the same time we do not want the job completion time to become excessively longer. From simulation results in Table 1-3, we conclude that the UBLBLB algorithm consumes low amount of energy almost comparable to the BF algorithm, its SLA violation ratio is much better than the BF and UBBF algorithms and almost similar to the UBLB algorithm.

CONCLUSION

In this study, we approached the scheduling problem in a data center as solving a static VM consolidation problem which was then modeled as a bin packing problem. The consolidation algorithm should try to minimize not only the total energy consumption in the data center but also the SLA violation ratio of submitted jobs. Moreover, we must consider more realistic situations when we deal with the consolidation algorithm as follow:

- The performance of a VM is affected by other VMs running in the same PM
- A PM can consist of multiple CPUs, each of which can have multiple cores with hyper-threading enabled
- VMs are created with pre-defined machine types

To enhance the performance further, a proper number of idle VMs are created for each machine type and this VM precreation is based upon the statistics of jobs including both inter-job arrival time and resource requirements of new jobs. With this idle VM precreation policy, there can be two situations when a new job arrives. The first case is when there are enough number of idle VMs of the proper machine type. In this case the VM consolidation problem becomes to which idle VMs the tasks of the new job should be allocated. The second case is when there not enough number of idle VMs and in this case the VM consolidation problem becomes on which PMs a necessary number of idle VMs should be created, so that, all the tasks of the new job can be assigned to VMs. The idle VM precreation policy brings about other issues and solved these issues too:

- After a new job is assigned to idle VMs, more idle VMs may need to be created to satisfy the idle VM precreation policy
- After a job is terminated, there may be more idle VMs than necessary and therefore, some of them need to be destroyed

In this study, we proposed four static VM consolidation algorithms considering all the issues explained:

- The best fit algorithm
- The upper-bounded best fit algorithm
- The upper-bounded load-balancing algorithm
- The upper-bounded and lower-bounded load-balancing algorithm

Basically each algorithm consists of three functions:

- Allocating tasks of a job to idle VMs
- Creating a new idle VM
- Destroying an unnecessary idle VMs

The performance of proposed algorithms are evaluated through simulation with synthetic workloads obtained by analyzing the characteristics of workloads in real data centers. We measured following three metrics:

- Ratio of idle PMs
- Service level agreement violation ratio
- The total energy consumption in a cloud

Analyzing collected measurement data, we conclude that the upper-bound and lower-bounded load-balancing algorithm achieves the low service level agreement violation ratio with the high ratio of idle PMs and the low amount of energy consumed.

ACKNOWLEDGEMENT

This research was supported by 2016 Hongik University Research Fund.

REFERENCES

Anonymous, 2019a. Amazon EC2 instance types. Amazon Web Services Inc., Seattle, Washington, USA. https://aws.amazon.com/ec2/instance-types/

Anonymous, 2019b. Machine types. Google LLC., USA. https://cloud.google.com/compute/docs/machine-types

Anonymous, 2019c. Sizes for Windows virtual machines in Azure. Microsoft, USA. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes

Armbrust, M., A. Fox, R. Griffith, A.D. Joseph and R. Katz *et al*., 2010. A view of cloud computing. Commun. ACM, 53: 50-58.

Barroso, L.A. and U. Holzle, 2007. The case for energy-proportional computing. Comput., 40: 33-37.

Beloglazov, A., J. Abawajy and R. Buyya, 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. Future Gen. Comput. Syst., 28: 755-768.

Glanz, J., 2012. Power, pollution and the internet. The New York Times, New York, USA. https://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html

Govindan, S., J. Liu, A. Kansal and A. Sivasubramaniam, 2011. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. Proceedings of the 2nd ACM Symposium on Cloud Computing, October 26-28, 2011, ACM, Cascais, Portugal, ISBN:978-1-4503-0976-9, pp: 1-22.

Koh, Y., R. Knauerhase, P. Brett, M. Bowman and Z. Wen *et al*., 2007. An analysis of performance interference effects in virtual environments. Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems & Software, April 25-27, 2007, IEEE, San Jose, California, USA., ISBN:1-4244-1081-9, pp: 200-209.

Lee, Y.C. and A.Y. Zomaya, 2012. Energy efficient utilization of resources in cloud computing systems. J. Supercomputing, 60: 268-280.

Li, A., X. Yang, S. Kandula and M. Zhang, 2010. CloudCmp: Comparing public cloud providers. Proceedings of the 10th Annual Conference on Internet Measurement, November 1-3, 2010, Melbourne, Australia, pp: 1-14.

Lin, C.C., P. Liu and J.J. Wu, 2011. Energy-efficient virtual machine provision algorithms for cloud systems. Proceedings of the 2011 4th IEEE International Conference on Utility and Cloud Computing, December 5-8, 2011, IEEE, Victoria, Australia, ISBN:978-1-4577-2116-8, pp: 81-88.

Lin, W., W. Wu and J.Z. Wang, 2016. A heuristic task scheduling algorithm for heterogeneous virtual clusters. Sci. Program., 2016: 1-10.

Lu, G. and W.H. Zeng, 2014. Cloud computing survey. Appl. Mech. Mater., 531: 650-661.

Mann, Z.A., 2015. Allocation of virtual machines in cloud data centers-a survey of problem models and optimization algorithms. ACM Comput. Surv., 48: 1-34.

Martello, S. and P. Toth, 1990. Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, ?Hoboken, New Jersey, USA., ISBN-13: 9780471924203, Pages: 296.

Meisner, D., B.T. Gold and T.F. Wenisch, 2009. PowerNap: Eliminating server idle power. Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV), March 07-11, 2009, ACM, Washington, DC., USA., ISBN:978-1-60558-406-5, pp: 205-216.

Mell, P. and T. Grance, 2011. The NIST definition of cloud computing. National Institute of Standards and Technology Special Publication, USA. http://scholar.googleusercontent.com/scholar?q=cache:xiGO0TJzMCsJ:scholar.google.com/+The+NIST+Definition+of+Cloud+Computing&hl=en&as_sdt=0,5

Mishra, A.K., J.L. Hellerstein, W. Cirne and C.R. Das, 2010. Towards characterizing cloud backend workloads: Insights from Google compute clusters. ACM SIGMETRICS Perform. Eval. Rev., 37: 34-41.

Mishra, M., A. Das, P. Kulkarni and A. Sahoo, 2012. Dynamic resource management using virtual machine migrations. IEEE. Commun. Mag., 50: 34-40.

Oh, F.Y.K., H.S. Kim, H. Eom and H.Y. Yeom, 2011. Enabling consolidation and scaling down to provide power management for cloud computing. Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'11), June 14-15, 2011, USENIX Association, Portland, Oregon, pp: 14-14.

Pu, X., L. Liu, Y. Mei, S. Sivathanu and Y. Koh *et al*., 2012. Who is your neighbor: Net i/o performance interference in virtualized clouds. IEEE. Trans. Serv. Comput., 6: 314-329.

Reiss, C., A. Tumanov, G.R. Ganger, R.H. Katz and M.A. Kozuch, 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. Proceedings of the 3rd ACM International Symposium on Cloud Computing (SoCC'12), October 14-17, 2012, ACM, San Jose, California, USA., ISBN:978-1-4503-1761-0, pp: 1-13.

Singh, S. and I. Chana, 2016. Cloud resource provisioning: Survey, status and future research directions. Knowl. Inf. Syst., 49: 1005-1069.

Verma, A., J. Bagrodia and V. Jaiswal, 2014. Virtual machine consolidation in the wild. Proceedings of the 15th International Conference on Middleware (Middleware'14), December 8-12, 2014, ACM, Bordeaux, France, ISBN:978-1-4503-2785-5, pp: 313-324.