Integrating Cloud Application Autoscaling with Dynamic VM Allocation

Michael Tighe and Michael Bauer
Department of Computer Science
The University of Western Ontario
London, Canada
{mtighe2|bauer}@csd.uwo.ca

***Abstract*—As the popularity of cloud computing continues to rise, more and more applications are being deployed in public clouds. To conserve provisioning costs while achieving performance objectives, clients should automatically scale up and down applications deployed in the cloud to match changing workload demands. The cloud provider, on the other hand, should attempt to consolidate load onto highly utilized physical machines, in order to reduce wasted power consumption. We propose a new algorithm combining both the automatic scaling of applications with dynamic allocation of virtual machines, in order to meet the goals of both the cloud client and provider. We evaluate this algorithm against running separate, independent autoscaling and dynamic allocation algorithms and show that the integrated algorithm can achieve better application performance with a significant reduction in virtual machine live migrations.**

I. Introduction

As the popularity of cloud computing continues to rise, more and more applications are being deployed in public clouds. Deploying an application in the cloud provides the application owners with near-instant access to resources, seemingly infinite scalability, and significantly reduced infrastructure management responsibilities. Public cloud offerings typically come in one of three flavours: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS). The target environment of this work is a virtualized IaaS cloud, which provides resources in the form of *virtual machines* (VMs) with low-level access to clients in an on-demand, pay-per-use manner. The main actors in this environment are the *provider*, the *client*, and the *user*. The cloud provider is the owner of the cloud infrastructure, which is rented to the clients. Clients deploy applications in the cloud, which are accessed by users. The provider provides services to the clients, and the clients provide services to their users. To take full advantage of the near-instant availability and scale of cloud resources, applications should be capable of dynamically scaling in size to meet demands. For example, web applications can experience highly variable loads, with peak demand far exceeding average. If enough resources are provisioned to meet peak demand, then a large amount of resource can be left underutilized, resulting in increased operating costs. Conversely, if the application is provisioned for average load, then performance can suffer during peak periods. Therefore, in order to reduce costs while meeting performance goals, applications must dynamically add and remove resources to match demand.

We consider an application as consisting of a number of interacting VMs which provide a single user-facing service. The automatic scaling of applications, known as autoscaling, can be achieved by dynamically adding or removing VMs to and from the application. For example, a web application may consist of a number of identical web servers, with incoming load balanced between them. When load increases, additional instances of the server can be added to the application. Conversely, when load decreases, instances can be removed. Some commercial implementations of autoscaling exist, such as Amazon Web Services' *Auto Scale* [1], which allows autoscaling to be triggered based on client defined rules.

The IaaS cloud provider also faces a set of challenges in order to make the best use of their infrastructure. A typical physical server can consume 50% to 70% of its peak power usage when idle [2], and as such, lightly utilized servers represent a significant waste of power. By consolidating load onto the fewest number of physical servers possible and switching idle servers into a power saving mode, a reduction in overall power usage and costs can be achieved. Resources such as CPU can be overcommitted; that is, a host can promise more resources to its set of hosted VMs than it actually possesses, with the assumption that not all VMs will require the resources simultaneously. This approach, however, can lead to resource contention when VMs require more resources than are available on the host, resulting in degraded application performance. A dynamic approach to VM allocation is therefore required. This can be accomplished through the use of VM live migration, in which running VMs can be moved from one host to another seamlessly, without incurring significant downtime. The problem of finding an optimal placement of VMs to hosts is known to be NP-hard [3]. As such, most work describing dynamic management approaches for large-scale systems makes use of first-fit heuristics [4] [5] [6] to periodically calculate new VM allocations in response to dynamic resource requirements. We refer to this process as *Dynamic VM Allocation*.

Most existing work looks at autoscaling applications or dynamic VM allocation separately. The goal of our work is to combine both into a single approach, leveraging control over autoscaling to assist in dynamic VM allocation, while simultaneously fulfilling the goals of both the cloud provider and the cloud client. From the perspective of the cloud client, the goal is to achieve their Service Level Agreement (SLA) with the lowest amount of resources required. The cloud provider, on the other hand, aims to reduce infrastructure use in order to conserve power and/or serve a larger number of clients, while ensuring that it does not sacrifice the performance of client VMs or affect autoscaling decisions of hosted applications. With these goals in mind, we present a new approach to handle both SLA-aware application autoscaling and dynamic VM allocation. Furthermore, we design our approach such that autoscaling decisions can remain under the control of the client, with the ability to potentially use alternative autoscaling algorithms, without modification to the main algorithm. The autoscaling algorithm makes the decision to scale considering only the best interests of the application itself, and our algorithm carries out these operations in a manner that helps achieve dynamic VM allocation goals.

We present a method for automatically scaling applications in Section III. In Section IV, we introduce dynamic VM allocation by way of an existing algorithm for VM consolidation, relocation and placement. Section V presents a new algorithm integrating autoscaling and dynamic VM allocation. We evaluate this algorithm against a static allocation, autoscaling alone, and running autoscaling and dynamic VM allocation algorithms independently in Section VI. Finally we conclude in Section VII.

II. Related Work

Existing related work can be considered in two categories: work on *autoscaling*, and work on *dynamic VM allocation*. Some work has focused on scaling the resource allocation of a single VM to meet the demands of applications running within it, but do not consider scaling applications out to multiple VMs [7]. Ghanbari et al. [8] examine existing alternative approaches to autoscaling, categorizing them into two categories: control theory approaches, and rule-based heuristics. They propose a control theory approach to scaling [9] an application running in a public IaaS cloud, which takes into consideration costs, constraints, and the characteristics of the IaaS environment. Their work is from the perspective of the client running applications within the cloud, with autoscaling logic running outside of the cloud. Our approach runs autoscaling algorithms within the cloud infrastructure itself. Chieu et al. [10] propose that the automatic scaling of application resources is a critical capability for clients deploying applications in the cloud. They present an autoscaling algorithm that falls into the rule-based category, scaling applications up or down based on the number of active user sessions in deployed web applications. Ferretti et al. [11] autoscale applications based on SLA achievement. They specify a threshold on response time as their SLA objective, and attempt to proactively scale the application based on continuous monitoring. Finally, Amazon Web Services provides an autoscale feature (AWS Auto Scale) [1], allowing clients to specify conditions under which additional VM instances should be added or removed. These conditions are rules based on monitored metrics, again falling into the rule-based heuristic category.

A great deal of work has been done in the area of dynamic VM allocation for the purpose of VM consolidation and thus power conservation. Early work looked at statically allocating VMs such that their resource demands would not be in conflict, based on the assumption that demands exhibit a periodic cycle. Approaches using a best-fit heuristic [12], linear programming [13] and vector bin packing [4] have been proposed. Other approaches have considered variable demand by periodically recalculating the entire VM allocation [14]. Work on truly dynamic VM allocation focuses mostly on first-fit [15] [16] or best-fit [6] heuristic solutions. Wood et al. [17] employ a first-fit decreasing heuristic that spreads load across hosts to reduce SLA violations. Keller et al. [5] studied several variations on a first-fit heuristic, concluding that the order in which VMs and hosts are considered for migration can have an impact on metrics such as power consumption and SLA violations. Some alternatives to best or first-fit heuristics, such as fuzzy logic-based controllers [18], have also be investigated.

Finally, there are a small number of works that examine something similar to combining autoscaling with dynamic VM allocation. Wuhib, Stadler and Spreitzer [19] propose an approach to distributed load-balancing in the cloud using a gossip protocol. Hosts balance workloads between them by adjusting load-balancing settings as well as by starting and stopping module (application) instances. The work was extended [20] to also consolidate workload for the purposes of reducing power consumption. The proposed solutions make use of a demand profiler to estimate resource requirements of modules and targets a Platform as a Service (PaaS) cloud, although the authors claim that the approach could be adapted to manage an IaaS cloud. Our approach does not require a demand profiler. Furthermore, their application scaling is performed with the goal of load balancing, rather than reducing the costs of clients running applications in the cloud. We focus on letting applications decide how they want to scale, and providing them with the resources the require. Jung et al. [21] perform application scaling and dynamic VM allocation using a layered queueing model of applications to determine the utility of configurations. They also require offline experimentation to determine the effects of configuration adaptation methods, which are later used to estimate effects online. Our approach does not require detailed models nor prior offline processing. Petrucci et al. [22] use a mixed integer programming model to optimize application placements. Unlike our work, they do not allow the client to determine auto-scaling conditions and the scalability of their approach is not fully evaluated. Finally, VDCPlanner [23] considers placing *virtual data centres* (i.e. sets of VMs with network link requirements), taking into account bandwidth requirements and making use of VM migration to improve placement. They also handle scaling requests and VM consolidation. They do not, however, consider dynamic VM resource requirements and overcommitting of resources.

Our work differs from the existing literature in that we focus on combining autoscaling with dynamic VM allocation into a single algorithm, capable of scaling applications while also consolidating load for power conservation. Other work considers only one of these goals, or targets a different environment.

III. Application Autoscaling

Applications are typically overprovisioned, resulting in significant underutilization of resources and increased cost [11]. By automatically scaling applications up and down to match changing workload demands, the client can provision, and pay for, only the resources it requires. We consider scaling resources by adding and removing VMs to and from the application, rather than by scaling resource allocations on individual hosts. We present a rule-based, heuristic autoscaling algorithm for use in our implementation, but it would be possible to make use of a more complex autoscaling algorithm without modification to our final integrated algorithm (Section V).

A. Application Model

In this work, we consider an *interactive* application, such as a three-tiered web application, in which a number of clients interact with the system, waiting for responses before issuing a new request. We implement a model of such an application in an open source simulator, DCSim [24] [25], for the purposes of evaluating our approach.

By our definition, an *Application* consists of one or more *Tasks*. Tasks are divided into one or more identical *Task Instances*, which share incoming workload via a *Load Balancer* component. The *size* of a task refers to the number of task instances in contains. Each task instance runs within a single Virtual Machine (VM), and each VM runs a single task instance. VMs can be placed on any host in the data centre. We model the application as a closed queueing network, and solve it using Mean Value Analysis (MVA). In addition to a set of tasks, an application has a specified *think time* for clients, and a *Workload* component. The workload defines the current number of clients using the application, which changes at discrete points in the simulation based on a trace file.

An individual task is defined by its *default* and *maximum size*, *service time* (time to process a single request), *visit ratio* (the average number of times each request must visit the task), and an expected amount of resources allocated to each task instance in order to achieve the specified service time (i.e. virtual cores, core speed, memory). Actual service time for each task instance is adjusted to reflect the effect of CPU contention on hosts, in the case that the CPU becomes overloaded.

B. Service Level Agreements

We evaluate the performance of an application deployed in the cloud based on achievement of its SLA. For our purposes, we define the SLA of an application in terms of an upper threshold on the response time of the entire application. When the response time exceeds this threshold, SLA is considered violated. Response time exceeding the threshold is a consequence of under provisioning CPU resources to a task instance(s) in the application, caused either by failing to scale up the application or by contention with other VMs on a host. We define *SLA Achievement* as the percentage of time during which SLA is not violated (higher is better), and report the average SLA achievement for the set of applications in our experiments.

C. Algorithm

We have implemented an autoscaling algorithm to perform the scaling up and down of applications based on current SLA metrics. We implement a rule-based, heuristic algorithm, as it is light-weight, does not require detailed knowledge or models of applications, and potentially allows application owners (cloud clients) to easily define the rules by which their application scales. Each application is monitored by its own manager, which executes Algorithm 1 on a regular schedule (e.g. every 5 minutes). Monitoring data from each task instance in the application is regularly sent to the manager. The algorithm requires the following as input: the application being scaled ($application$); the time (t); window sizes for sliding average response time (r_W) and CPU utilization (u_W); an *SLA Threshold* (T_{SLA}) on the response time of the application; an *SLA Warning* value (W_{SLA}), specified as a percentage of the SLA response time threshold; and a *CPU Safe* value (S_{CPU}). Function $\overline{R(application, window)}$ calculates the average response time of an application over a sliding window. We calculate the average response time (r) over the window r_W (line 1), which is then compared against the warning value (line 2). If it exceeds this value, a *scale up* operation is performed to add additional VMs to the application. The algorithm then iterates through the tasks that comprise the application (lines 4-8), and calculates the change in response time (r_Δ) since the last execution of the algorithm. Function $R(task, time)$ returns the response time of a task at a specified time. The algorithm chooses the task with the largest increase in response time, and whose size ($|task|$) has not reached its maximum size ($size_M[task]$). A new instance is then added to this task (line 8).

In the case that the response time is below the warning level, then the algorithm looks for a task on which to perform a *scale down* operation. Scaling down is performed based on CPU utilization, rather than SLA. Function $\overline{U(instance, window)}$ computes the average CPU utilization of a task instance over a sliding window. The total CPU utilization of all instances in the task ($task_\theta$), averaged over a sliding window of size u_W (line 13), is computed and divided by task size minus one to estimate the utilization of remaining task instances, should one be removed (line 14). If this value falls below a specified *CPU Safe* value (S_{CPU}), then the task is a candidate for a *scale down* operation. The algorithm chooses the candidate task with the lowest utilization from which to remove an instance (lines 15-16), and removes an arbitrary instance from the task (line 17).

The threshold values for SLA Warning and CPU Safe, as well as the metric to evaluate (e.g. response time, throughput, CPU utilization) could easily be defined by the cloud client who owns the application, similar to rules defined in Amazon Web Services Auto Scale [1]. This gives the client some level of control over decisions that can impact both their SLA achievement as well as their operating costs.

Algorithm 1: Autoscaling Algorithm

```
Data: $\underline{application,\ t,\ r_W}$, $u_W$, $T_{SLA}$, $W_{SLA}$, $S_{CPU}$
1  $r \leftarrow \overline{R(application, r_W)}$ ; $target \leftarrow$ NIL
2  if $r > T_{SLA} \times W_{SLA}$ then
3      $target_\triangle \leftarrow 0$
4      for $task \in tasks[application]$ do
5          $r_\triangle \leftarrow R(task, t) - R(task, t-1)$
6          if $r_\triangle > target_\triangle$ & $|task| < size_M[task]$ then
7              $target_\triangle \leftarrow r_\triangle$ ; $target \leftarrow task$
8      if $target \neq NIL$ then addInstance($target$)
9  else
10     $target_\theta \leftarrow \infty$
11     for $task \in tasks[application]$ do
12         $I \leftarrow instances[task]$
13         $task_\theta \leftarrow \sum_{i \in I} \overline{U(i, u_W)}$
14         if $size[task] > 1$ & $\dfrac{task_\theta}{|task| - 1} \leqslant S_{CPU}$ then
15             if $task_\theta < target_\theta$ then
16                 $target_\theta \leftarrow task_\theta$ ; $target \leftarrow task$
17     if $target \neq NIL$ then removeInstance($target$)
```

IV. Dynamic VM Allocation

In order for the cloud provider to make the best use of their infrastructure, VMs should be consolidated on as few hosts (physical machines) as possible, while still providing the resources they require to meet their performance objectives. This can be accomplished by overcommitting resources, such as CPU, and dynamically reallocating VMs to match current demand situations. When a host is running low on resources, a VM can be moved to another host through a *live migration* operation. In our previous work, we present a solution based on a first-fit heuristic [5], which we will use as the basis for our work here, as it is representative of a good deal of the related work in the area. The solution classifies hosts into four categories: *stressed* hosts (S) have a CPU utilization higher than a specified $upper_{CPU}$ threshold value; *partially utilized* hosts (P) have a CPU utilization below $upper_{CPU}$ and above a lower threshold, $lower_{CPU}$; *underutilized* hosts (L) have a CPU utilization below $lower_{CPU}$; and *empty* hosts (E) are not hosting any VMs, and may be in a low power state such as *suspended* or *off*. Each category is sorted by CPU utilization in either ascending or descending order to control the behaviour of the algorithm. We refer the reader to previous work [5] [26] [27] for more details.

The approach consists of three basic operations:

1) VM Relocation: When a host is *stressed* (CPU utilization exceeding $upper_{CPU}$), one of its VMs must be relocated (migrated) to another host to relieve the stress situation. VM Relocation is triggered on a regular interval, every 10 minutes [26]. Algorithm 2 gives an overview of the relocation algorithm, which takes the set of hosts in the data centre ($hosts$) as input. First, hosts are classified as described above (line 1). The *stressed* hosts are used as migration sources (line 2), and the concatenation of the remaining categories, sorted, is used as the target list (line 3). For each source host (h_s), its set of VMs (vms) is sorted and then iterated, searching for a target host (h_t) for each VM (v) (lines 4-13). As soon as a suitable VM and target is found, the algorithm triggers the migration operation and continues to the next source host.

Algorithm 2: VM Relocation

```
Data: $hosts$
1  $S, P, L, E \leftarrow$ classify($hosts$)
2  $sources \leftarrow$ sort($S$)
3  $targets \leftarrow$ sort($P$) $\cdot$ sort($L$) $\cdot$ sort($E$)
4  for $h_s \in sources$ do
5      $vms \leftarrow$ sort($vms[h_s]$)
6      $success \leftarrow$ FALSE
7      for $v \in vms$ do
8          for $h_t \in targets$ do
9              if hasCapacity($h_t$, $v$) then
10                 migrate($h_s$, $v$, $h_t$)
11                 $success \leftarrow$ TRUE
12                 break
13         if $success$ then break
```

2) VM Consolidation: The VM Consolidation operation attempts to consolidate VMs on as few hosts as possible, and is triggered once every hour [26]. When a host is *underutilized* (CPU less than $lower_{CPU}$), it is a candidate for being shut down (or suspended) to conserve power. This is accomplished by first finding migration targets for each VM on the candidate host, performing the migrations, and switching the host into a lower power state. The algorithm is similar to that of VM Relocation (Algorithm 2), except that *underutilized* hosts are used as *sources*, and any non-*stressed* host can be used as a target.

3) VM Placement: VM Placement is executed each time a new request to create a VM arrives at the data centre. It is responsible for selecting an appropriate host for the new VM, and instantiating it. Again, the algorithm works similarly to the VM Relocation algorithm (Algorithm 2).

V. Integrating Autoscaling and Dynamic VM Allocation

We now present a single algorithm integrating both autoscaling (see Section III) and dynamic VM allocation (see Section IV). The decision to perform an application *scale up* or *scale down* operation is still left to the application. However, by leveraging the knowledge of which applications are scaling and by controlling which task instances are removed and where new instances are placed, we can aide VM Relocation and Consolidation and reduce the number of required migrations.

The *VM Relocation* and *VM Consolidation* operations of dynamic VM allocation are combined with the *autoscaling* algorithm, while the *VM Placement* operation is handled separately. Algorithm 3 presents a portion of the integrated algorithm, dealing with stressed hosts (i.e. VM Relocation). It requires as input the set of hosts in the data centre ($hosts$), the set of applications ($applications$) running in the data centre, and a timeout on host stress (N_S). Hosts are first classified as presented in Section IV. We then record the number of consecutive times the host has been *stressed* ($n_S[h]$) in lines 2-3. In line 4, the *evaluateScaling()* method executes a minor variation on Algorithm 1, which simply returns a list of tasks to be scaled up (T_{UP}) and scaled down (T_{DOWN}), rather than performing the operations directly. Note that this algorithm could easily be exchanged with any autoscaling algorithm desired by the application owner, so long as it returns a set of tasks to scale up and down. Furthermore, it is not required that each application use the same autoscaling algorithm. For our current implementation, however, the algorithm presented in Section III is used.

The algorithm then iterates through *stressed* hosts (line 5). Line 7 calculates the amount by which CPU utilization is over the stress threshold ($over_{CPU}$). We then attempt to determine whether the effect of pending *scale up* operations will relieve the stress situation without any further action. Task instances of a single task share incoming work, and as such, a *scale up* operation results in an additional task instance (VM) taking a portion of the workload from the existing task instances. We calculate the total estimated reduction in CPU for the host (Γ_{CPU}), and determine if it will be enough to return the host to a non-*stressed* state (lines 8 - 13). If so, no further action is taken.

Next, the algorithm looks for pending *scale down* operations in which the task being scaled down has an instance on the *stressed* host (lines 14-22). If such a task is found, then the instance on the *stressed* host is removed (line 18), and the task is removed from the list of pending *scale down* operations (line 19). No further action is taken. Finally, if neither of the previous steps were successful, and the host has been *stressed* for a specified timeout period (N_S), then we trigger relocation via migration. The timeout value is set to 2 based on an evaluation of multiple values. Relocation is performed in a similar manner to Algorithm 2.

The remainder of the algorithm, omitted due to space considerations, continues to make use of pending *scale up* and *scale down* operations to avoid migrations. First, *scale down* operations in which an instance of the task is located on a host with CPU utilization within 5% of the $upper_{CPU}$ threshold are performed, with the first instance located on such a host chosen as the instance to be removed. This is done to reduce the likelihood of hosts becoming *stressed* in the future. Next,

Algorithm 3: Integrated Algorithm - Stress Handling

```
   Data: $hosts$, $applications$, $N_S$
 1 $S, P, L, E \leftarrow$ classify($hosts$)
 2 for $h \in S$ do $n_S[h] \leftarrow n_S[h] + 1$
 3 for $h \in P \cdot L \cdot E$ do $n_S[h] \leftarrow 0$
 4 $T_{UP}, T_{DOWN} \leftarrow$ evaluateScaling($applications$)
 5 for $h \in S$ do
 6     $done \leftarrow$ FALSE
 7     $over_{CPU} \leftarrow U[host] - upper_{CPU}$
 8     $\Gamma_{CPU} \leftarrow 0$
 9     for $task \in T_{UP}$ do
10         for $i \in instances[task]$ do
11             if $i \in instances[h]$ then
12                 $\Gamma_{CPU} \leftarrow \Gamma_{CPU}$ + estimateReduction($i$)
13     if $\Gamma_{CPU} \geqslant over_{CPU}$ then $done \leftarrow$ TRUE
14     if $done$ = FALSE then
15         for $task \in T_{DOWN}$ do
16             for $i \in instances[task]$ do
17                 if $i \in instances[h]$ then
18                     removeInstance($task$, $i$)
19                     $T_{DOWN} \leftarrow T_{DOWN} \setminus \{task\}$
20                     $done \leftarrow$ TRUE
21                     break
22             if $done$ then break
23     if $done$ = FALSE & $n_S[h] \geqslant N_S$ then relocate($h$)
```

scale up operations are performed, placing new instances first on *partially-utilized* hosts, and then on *underutilized* hosts.

The next step is to attempt to consolidate load. We do so in a similar manner to the *VM Consolidation* operation described in Section IV, except that we attempt to leverage remaining *scale down* operations that have task instances on underutilized hosts. We first calculate a $shutdownCost$ for shutting down each *underutilized* host. The $shutdownCost$ is defined as the number of VMs that must be migrated to shut down the host, given that any tasks pending a *scale down* operation with instances on the candidate host will choose those instances to remove. For each remaining *scale down* operation, we select for removal the task instance on the *underutilized* host with the lowest $shutdownCost$. If no task instances are located on an *underutilized* host, then the task instance on the host with the lowest CPU utilization is chosen. Finally, we attempt to migrate away any remaining VMs and shut down *underutilized* hosts that have been underutilized for a specified timeout period. The timeout value is set to 12 based on an evaluation of multiple values. Essentially, we are intelligently selecting *scale down* operations such that we remove instances from only a small set of hosts, and increase the likelihood of successfully shutting down an *underutilized* host with fewer (or zero) migrations required.

VI. Evaluation

In this section, we will evaluate our approach by conducting a set of experiments using a simulation tool, DCSim [24] [25]. We begin by evaluating our autoscaling algorithm by comparing it with a static allocation for peak application demand. Next, we examine autoscaling alongside an existing dynamic VM allocation algorithm from our previous work [5] [26] (see Section IV). Finally, we evaluate our new, integrated autoscaling and dynamic VM allocation algorithm, and compare it to running separate algorithms, autoscaling alone, and static allocation.

A. Metrics

*Active Hosts (***Hosts***):* The average number of hosts in the *On* state. The higher the value, the more physical hosts are being used to run the workload.

*Average Active Host Utilization (***Host Util.***):* The average CPU utilization of all hosts in the *On* state. The higher the value, the more efficiently resources are being used.

*Power Consumption (***Power***):* Power consumption is calculated for each host, and the total kilowatt-hours consumed during the simulation are reported. Power consumption is calculated using results from the SPECPower benchmark [28], and is based on CPU utilization.

*Application Size (***Size***):* Application Size is the number of VMs belonging to the application. We report the average size as a percentage of the maximum size defined for each application.

*SLA Achievement (***SLA***):* SLA Achievement is the percentage of time in which the SLA conditions are met. See Section III-B for details.

*Autoscaling Operations (***AS Ops***):* The number of *scale up* and *scale down* operations executed.

*Number of Migrations (***Migrations***):* The number of migrations triggered during the simulation. Typically, a lower value is more desirable, as less bandwidth would be used for VM migrations.

B. Experimental Setup

Our simulated data centre consists of 200 host machines based on the HP ProLiant DL160G5, with 2 quad-core 2.5GHz CPUs and 16GB of memory. Hosts are modelled to use a work-conserving CPU scheduler, as available in major virtualization technologies. As such, CPU shares that are not used by one VM can be used by another. No maximum cap on CPU is set for VMs. In the event that the CPU is at maximum capacity and VMs must compete for resources, VMs are assigned CPU shares in a fair-share manner. Memory is statically allocated and is *not* overcommitted.

We model a set of interactive applications running within the data centre, as described in Section III-A. Task instances run on VMs with 1 virtual core and 1GB of RAM. We defined a set of 4 artificial applications of varying configurations, described in Table I. The *default size* refers to the default number of instances created for each task, although the minimum is 1 task instance per task. The maximum size for each application is calculated as its default size scaled up by a *scale factor* of $[3, 6]$, chosen randomly. The user think time for each application is set at 4 seconds.

App.	Task	Service Time (s)	Visit Ratio	Default Size
1	1	0.005	1	1
	2	0.02	1	4
	3	0.01	1	2
2	1	0.005	1	1
	2	0.02	1	4
3	1	0.005	1	1
	2	0.02	1	4
	3	0.01	1	2
	4	0.01	0.5	1
	5	0.02	0.5	2
4	1	0.01	1	1

TABLE I
Applications

Alg.	Hosts	Power	SLA	Size	AS Ops
Static	194.7	4177kWh	100.0%	100%	N/A
Autoscaling	147.7	3424kWh	95.8%	64%	9503

TABLE II
Autoscaling

The number of active users for each application changes dynamically based on real workload traces. Each application uses a trace built from one of 3 sources: *ClarkNet*, *EPA*, and *SDSC* [29]. We compute a normalized workload level based on request rate, in 100 second intervals, for each trace. These levels are used to define the current number of users of each application. The normalized workloads are scaled such that the peak number of clients receive a response time of 0.9 seconds (just under the defined SLA threshold) when all application tasks have their maximum number of instances. To ensure that applications do not exhibit synchronized behaviour, each applications starts its trace at a randomly selected offset time.

A set of 50 applications is generated for each experiment, with randomly chosen configurations and scale factors. Applications are placed at the start of the simulation, and run for the duration of the experiment. We use 10 different randomly generated application sets to evaluate our work, and all presented results are averaged across experiments using these 10 sets. Simulations run for 6 days, and we discard data from the first day of simulation to allow the system to stabilize before recording results, resulting in 5 days of results.

C. Autoscaling

We compared the autoscaling algorithm, defined in Section 1, against a static allocation. The static allocation creates the maximum number of instances for each application task, places all task instances (VMs) at the start of the experiment, and does not modify the allocation or number of instances further. As such, each application is allocated enough resources to meet peak demand, at all times. In contrast, the autoscaling algorithm attempts to adapt the size of each application to match fluctuating workload demands. Tuning parameters, such as W_{SLA}, S_{CPU}, and r_W, were chosen based on an evaluation of a large number of potential values, the results of which have been omitted for space considerations. The autoscaling algorithm is executed every 5 minutes, again based on a evaluation of a number of possible frequencies. New task instances from *scale up* operations are placed in the first available host. During a *scale down* operation, an arbitrary task instance is chosen, as the autoscaling algorithm is based on potentially client-defined rules and has no knowledge of host state.

Table II presents the results of the autoscaling evaluation. The autoscaling algorithm results in an 18% reduction in power consumption, with average application size at 64% of maximum. This represents a considerable saving in terms of resource provisioning costs for clients, with a reasonable loss in SLA performance (average SLA achievement for all applications is 95.8%).

D. Autoscaling with Dynamic VM Allocation

We now add an existing dynamic VM allocation approach, running alongside our autoscaling algorithm. They run independently, with no knowledge of the actions being performed by the other. The autoscaling algorithm has knowledge only of application metrics, such as response time and VM CPU utilization, and makes decisions accordingly. The dynamic VM allocation algorithm has knowledge of host states, but no knowledge or control over application metrics and operations. New task instances created by the autoscaling algorithm are, however, placed using the VM Placement operation in Section IV. We run a set of 9 different experiments with varying values of $upper_{CPU}$ and $lower_{CPU}$. Tuning parameters for the autoscaling algorithm are identical to the values used in Section VI-C.

Table III presents the results of our experiments. Compared to the autoscaling algorithm alone (Table II), we see a 37-43% savings in power consumption. This comes at the expense of SLA achievement, which for all by the lowest $upper_{CPU}$ values falls below 90%. While lower $upper_{CPU}$ values lead to better SLA achievement, they also lead to a significant increase in the number of migrations performed. In addition, there is an increase seen in the number of autoscaling operations, which indicates that consolidation is affecting the behaviour of the autoscaling algorithm. We argue that using $upper_{CPU} = 80$ and $lower_{CPU} = 60$ is the best choice of configuration for the separate dynamic VM allocation algorithm, as it provides significant power savings while maintaining at least 90% SLA achievement. We will use this configuration for further comparison in Section VI-F.

E. Integrated Algorithm

Finally, we evaluate our newly developed integrated autoscaling and dynamic VM allocation algorithm. Once again, we run a set of 9 different experiments with varying values of $upper_{CPU}$ and $lower_{CPU}$. Autoscaling decisions are made in the *evaluateScaling()* method (see Algorithm 3) using the same autoscaling tuning parameter values used in Section VI-C, to provide a fair comparison.

Table IV presents the results of our experiments with the integrated algorithm. Compared with separate autoscaling and dynamic VM allocation algorithms, we can see a notable improvement in SLA achievement, reaching nearly that of autoscaling alone. Furthermore, we note a significant decrease in the number of migrations, falling in line with our goals for the development of the algorithm, as well as fewer autoscaling operations. We argue that using $upper_{CPU} = 90$ and $lower_{CPU} = 40$ is the best choice of configuration for the integrated algorithm, as it provides significant power savings, sacrifices very little in terms of SLA achievement, and performs the fewest number of migrations. We will use this configuration for further comparison in Section VI-F.

F. Discussion

Table V presents selected results from our experiments with a simulated data centre. The static allocation of peak resource requirements for all applications predictably leads to 100% SLA achievement, but high power consumption. Enabling autoscaling results in a reduction in power consumption, and as discussed in Section VI-C, a reduction in provisioning costs for the client. This comes at the expense of a slight loss in SLA achievement.

Enabling dynamic VM allocation, with autoscaling running separately and independently, provides a further savings in power consumption. This comes at the expense of SLA achievement, and a large number of migrations. Finally, by integrating autoscaling and dynamic VM allocation into a single algorithm, we see a return of SLA achievement and a significant reduction in required migrations. This comes at the expense of some of the power savings gained by running separate algorithms, but still represents a large savings when compared to autoscaling alone. Dynamic VM allocation also seems to slightly increase the number of autoscaling operations triggered, likely due to CPU contention affecting application response time and triggering *scale up* operations. While the effect is likely not be enough to warrant concern, future work could investigate methods to avoid this situation.

VII. Conclusions and Future Work

In order to reduce costs and infrastructure requirements, both cloud clients and providers must strive to make efficient use of their resources. From the client perspective, this entails provisioning only the resources it requires in order to meet its objectives. For the cloud provider, this means consolidating load on as few physical machines as possible, without affecting client application performance. We introduced an autoscaling algorithm which automatically scales applications to meet current demands and maintain SLA achievement. We then ran our autoscaling algorithm alongside a dynamic VM allocation algorithm from our previous work [5] [26] to perform consolidation. This allows the cloud provider to make better use of their infrastructure, and reduce power consumption. While power consumption was reduced, it was at the expense of SLA achievement and a large number of migrations. Finally, we developed an integrated algorithm performing both autoscaling and dynamic VM allocation. The algorithm still allows applications to decide whether or not to perform a scaling operation, but intelligently selects instances

$upper_{CPU}$	$lower_{CPU}$	Hosts	Host Util.	Power	SLA	AS Ops	Migrations
90	60	75.5	76.0%	1957kWh	88.6%	12121	11002
90	50	77.2	74.2%	1989kWh	88.5%	12098	10106
90	40	80.1	71.6%	2041kWh	88.6%	12073	9362
85	60	77.8	74.3%	2008kWh	89.6%	12023	16546
85	50	80.1	72.3%	2049kWh	89.6%	12022	15477
85	40	82.9	69.8%	2098kWh	89.7%	11998	14296
80	**60**	**80.6**	**72.5%**	**2066kWh**	**90.7%**	**11895**	**21834**
80	50	83.2	70.3%	2115kWh	90.7%	11876	20411
80	40	86.2	67.9%	2167kWh	90.7%	11875	19018

TABLE III
SEPARATE AUTOSCALING AND ALLOCATION EXPERIMENTS

$upper_{CPU}$	$lower_{CPU}$	Hosts	Host Util.	Power	SLA	AS Ops	Migrations
90	60	73.9	76.8%	1923kWh	87.2%	11972	8602
90	50	88.1	67.9%	2219kWh	93.1%	11386	5393
90	**40**	**100.7**	**60.7%**	**2460kWh**	**95.3%**	**10320**	**3778**
85	60	74.8	76.4%	1946kWh	88.2%	11902	8703
85	50	88.7	67.9%	2232kWh	93.6%	11293	5619
85	40	100.6	60.8%	2459kWh	95.4%	10283	4193
80	60	77.0	75.2%	1994kWh	89.6%	11791	8887
80	50	89.2	67.6%	2245kWh	93.9%	11229	6013
80	40	100.2	61.1%	2453kWh	95.4%	10253	4612

TABLE IV
INTEGRATED ALGORITHM EXPERIMENTS

Alg.	Hosts	Power	SLA	AS Ops	Migrations
Static	194.7	4177kWh	100.0%	N/A	N/A
Autoscaling	147.7	3424kWh	95.8%	9503	N/A
Separate	80.6	2066kWh	90.7%	11895	21834
Integrated	100.7	2460kWh	95.3%	10320	3778

TABLE V
ALGORITHM COMPARISON

for removal and placements for new instances to aide in dynamic allocation. This approach was able to restore much of the lost SLA from the approach using separate algorithms, while greatly reducing the number of migrations required.

Future work includes considering placement constraints on application tasks and task instances, considering the network topology of the data centre and minimizing network usage between communicating VMs. A more intelligent autoscaling algorithm could also be investigated. Finally, we have previously investigated a distributed algorithm for dynamic VM allocation [27], which could be extended to design a distributed application autoscaling and dynamic allocation algorithm.

REFERENCES

[1] (2013) Amazon EC2 Auto Scale. Amazon Web Services, Inc. [Online]. Available: http://aws.amazon.com/autoscaling/

[2] L. Barroso and U. Holzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec 2007.

[3] C. Hyser, B. Mckee, R. Gardner, and B. J. Watson, "Autonomic virtual machine placement in the data center," HP Laboratories, Tech. Rep. HPL-2007-189, Dec. 2007.

[4] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *J. Parallel Distrib. Comput.*, vol. 70, no. 9, pp. 962–974, Sep. 2010.

[5] G. Keller, M. Tighe, H. Lutfiyya, and M. Bauer, "An analysis of first fit heuristics for the virtual machine relocation problem," in *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, Oct. 2012.

[6] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency Computat.: Pract. Exper.*, pp. 1–24, 2011.

[7] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 5.

[8] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 716–723.

[9] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a iaas cloud," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 173–178.

[10] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *e-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*. IEEE, 2009, pp. 281–286.

[11] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini, "Qos–aware clouds," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 321–328.

[12] M. Cardosa, M. R. Korupolu, and A. Singh, "Shares and utilities based power consolidation in virtualized server environments," in *IM Proceedings, 2009 IEEE/IFIP Int. Symp. on*, 2009.

[13] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE TSC*, vol. 3, no. 4, pp. 266 –278, 2010.

[14] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *IM Proceedings, 2007 IEEE/IFIP Int. Symp. on*, 2007, pp. 119–128.

[15] G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," in *NOMS Proceedings, 2006 IEEE/IFIP*, 2006.

[16] A. Verma, P. Ahuja, and A. Neogi, "pmapper: power and migration cost aware application placement in virtualized systems," in *Proceedings of the 9th ACM/IFIP/USENIX Int. Conf. on Middleware*, 2008.

[17] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *NSDI Proceedings, 4th Symp. on*, Cambridge, MA, USA, Apr. 2007, pp. 229–242.

[18] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and

A. Kemper, “An integrated approach to resource pool management: Policies, efficiency and quality metrics,” in *38th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, June 2008.

[19] F. Wuhib, R. Stadler, and M. Spreitzer, “Gossip-based resource management for cloud environments,” in *Network and Service Management (CNSM), 2010 International Conference on*. IEEE, 2010, pp. 1–8.

[20] R. Yanggratoke, F. Wuhib, and R. Stadler, “Gossip-based resource allocation for green computing in large clouds,” in *Network and Service Management (CNSM), 2011 7th Int. Conf. on*, Oct. 2011.

[21] 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*. Springer, 2009, pp. 163–183.

[22] V. Petrucci, E. V. Carrera, O. Loques, J. C. Leite, and D. Mossé, “Optimized management of power and performance for virtualized heterogeneous server clusters,” in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE, 2011, pp. 23–32.

[23] M. F. Zhani, Q. Zhang, G. Simon, and R. Boutaba, “Vdc planner: Dynamic migration-aware virtual data center embedding for clouds.”

[24] M. Tighe, G. Keller, M. Bauer, and H. Lutfiyya, “Towards an improved data centre simulation with DCSim,” in *SVM Proceedings, 7th Int. DMTF Academic Alliance Workshop on*, Oct. 2013.

[25] DCSim on GitHub. [Online]. Available: https://github.com/digs-uwo/dcsim

[26] G. Foster, G. Keller, M. Tighe, H. Lutfiyya, and M. Bauer, “The Right Tool for the Job: Switching data centre management strategies at runtime,” in *Integrated Network Management (IM), 2013 IFIP/IEEE International Symposium on*, May 2013.

[27] M. Tighe, G. Keller, H. Lutfiyya, and M. Bauer, “A Distributed Approach to Dynamic VM Management,” in *Network and Service Management (CNSM), 2013 9th International Conference on*. IEEE, 2013.

[28] (2013, May) Specpower_ssj2008 benchmark. Standard Performance Evaluation Corporation. [Online]. Available: http://www.spec.org/power_ssj2008/

[29] (2013, May) The internet traffic archive. [Online]. Available: http://ita.ee.lbl.gov/