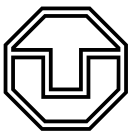


TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Professorship of Systems Engineering

TOWARDS AUTO-SCALING IN THE CLOUD: ONLINE RESOURCE ALLOCATION TECHNIQUES

Lenar Yazdanov
Born on: 25th February 1986 in Brezhnev, USSR

DISSERTATION

to achieve the academic degree

DOCTOR OF PHILOSOPHY (PH.D.)

First referee
Prof. Christof Fetzer, Ph.D.
Technische Universität Dresden, Germany

Second referee
Prof. Rüdiger Kapitza, Ph.D.
Technische Universität Braunschweig, Germany

Submitted on: 30th September 2015
Defended on: 26th September 2016

ABSTRACT

Cloud computing provides an easy access to computing resources. Customers can acquire and release resources any time. However, it is not trivial to determine *when* and *how many* resources to allocate. Many applications running in the cloud face workload changes that affect their resource demand. The first thought is to plan capacity either for the average load or for the peak load. In the first case there is less cost incurred, but performance will be affected if the peak load occurs. The second case leads to money wastage, since resources will remain underutilized most of the time. Therefore there is a need for a more sophisticated resource provisioning techniques that can automatically scale the application resources according to workload demand and performance constrains.

Large cloud providers such as Amazon, Microsoft, RightScale provide auto-scaling services. However, without the proper configuration and testing such services can do more harm than good. In this work I investigate application specific online resource allocation techniques that allow to dynamically adapt to incoming workload, minimize the cost of virtual resources and meet user-specified performance objectives.

ACKNOWLEDGEMENTS

First, I would like to thank Prof. Christof Fetzer for giving me opportunity to join Systems Engineering group. I have learned a lot from him. He always motivated me and provided many ideas. I also want to thank Prof. Rüdiger Kapitza for serving as the thesis external reviewer. He gave me some valuable hints when I start writing the thesis.

I would like to thank the people with whom I discussed my papers and who have commented on early versions of the papers: Diogo Behrens, Stefan Weigert, Andre Martin, Do Le Quoc.

My many thanks to Martin Nowack who helped me to setup experimental environment in the beginning and Stephan Creutz who introduced me to the wonderful world of Linux.

I also want to thank my colleges Robert Krahn and Thomas Knauth for reviewing on early drafts of the thesis and correcting my English mistakes.

My especial thanks go to my wife Dinara who supported me during my PhD. Without her I would not have time and strength to write the thesis. And of course, I want to thank my parents who were helping me and my family over the last five years.

CONTENTS

INTRODUCTION

1.1 MOTIVATION

Cloud computing lowered the barrier of entry to an infinite amount of computing resources. Therefore, nowadays any person in the world can rent computing resources to run an application. Usually the resources are delivered in the form of virtual machines (VMs). In comparison to traditional provisioning techniques that require upfront servers' deployment, cloud users can acquire and release resources on-demand. However, it is not easy to answer the question about *when* to allocate and *how many* resources to allocate.

To identify the right amount of resources to lease the user needs to consider a number of factors: such as application elasticity, workload dynamics, user-defined performance objectives and conversion of the performance objective to resource allocation. For a non-expert cloud user that has limited knowledge about the application and its resource demand pattern, it is hard to make an optimal scaling decision.

Cloud market offers a variety of resource allocation schemes. The user can choose a VM from the set of predefined templates or specify a VM he needs. Later on, during runtime it is possible to change the application resource capacity by modifying the number of VMs dedicated to the application (*horizontal scaling*) or adapt individual VM resources (*vertical scaling*). The number of possible resource allocation strategies becomes too large to be managed by a human. Therefore there is a need for automating the process of resource allocation. Auto-scaling services offered by cloud providers simplify the process of acquiring and releasing resources, but leave a burden of scaling policy design to the user.

The focus of this thesis is techniques and approaches for online scaling policy discovery. To design optimal scaling policy the user has to address a number of challenges. These challenges motivate our research. In this work we present resource allocation controllers that perform horizontal and vertical resource scaling. There are two objectives that we target. First, the amount of assigned resources should be enough to meet the application performance objectives. Second, the cost of running the application should be minimal.

1.2 THESIS OUTLINE

This thesis divided in to 8 chapters including introduction. The following paragraphs give a short overview of each chapter.

Chapter 2 gives an overview of resource scaling types, applications and trends in cloud market. Over the last decade cloud became a popular computing platform. In contrast to traditional provisioning approaches it offers flexibility in terms amount computational resources that user can acquire and the resources rental time. Users can modify the capacity of the application during runtime either by scaling it horizontally or vertically. The choice of the method depends on the application type, expected resource demand and utilization of the application. It is common today that datacenters host mix of applications. In general there are two types of applications. The first type consists of latency-sensitive interactive applications. To the second type belong resource intensive batch applications. Each application type requires an individual provisioning strategy. Cloud market is not static, it constantly evolves. Economic interests of cloud users force the providers to shift towards VM customization model and shorter billing cycles. In contrast to fixed size VM allocation model, the observed trends in the market require techniques for dynamic fine-granular resource allocation.

Chapter 3 presents the process of auto-scaling system design and gives an overview of techniques for automating resource allocation. The process of auto-scaling consists of four phases: monitoring, analysis, planning and execution. Analysis and planning constitute the core of the system. The analysis phase determines the current state of the application or predicts future needs. Once the state is known (or predicted) the auto-scaler plans how to perform resource allocation. The scaling decisions are made by exploiting the application performance model.

The model describes relationships between a set of parameters such as the application performance, incoming workload, assigned amount of virtual resources and service level objectives. The model can be defined offline by an expert user over the set of system identification experiments. Alternatively a hybrid and an online model discovery approaches can be used. The last two offer flexibility in terms of the scaling policy adaptation. There is a number of techniques that are used to describe the model. Generally, they are classified in five groups: threshold based, reinforcement learning, queuing theory, time series analysis and control theory. In *Chapter 3* we present each technique, discuss its advantages and disadvantages, and review related work.

Chapter 4 describes an approach for collocation of VMs that support vertical scaling. Many interactive applications have varying resource demand. Running these applications in a fixed size VM leads to resource wastage. In contrast to fixed size VM model, VM reconfiguration allows to follow resource demand curve of interactive applications. However, VM collocation is a challenging task for cloud providers that want to support vertical scaling. It is hard to give performance guarantees for the applications running in collocation with a VM that can be dynamically reconfigured. One would need to maintain a certain resource headroom. However, it will lead to low utilization of provider's infrastructure. To solve the problem we propose to collocate interactive and batch applications. Batch applications tolerant to performance slowdown and can make progress even under resource pressure. In *Chapter 4* we design time-series based auto-scaling controller that follows the interactive application resource demand and assigns remaining resources to the batch application. From one side the controller delivers a high performance to the interactive application, from another side it allows the batch application to harvest the residual resources and improve the provider's infrastructure utilization.

In *Chapter 5* we design autonomic resource scaling controller. The controller exploits reinforcement learning approach. Reinforcement learning is a knowledge-free approach for the application performance model identification. It allows to adapt scaling decisions with respect to changing resource demand. Cloud user does not need to define particular model for the application. However, the drawback of RL approaches is a significant learning time. Therefore, auto-scaling systems that use RL have bad performance in early steps. *Chapter 5* describes a technique that allows significantly improve the learning time. Typically, the reinforcement learning agent after each action updates only one state-action transition. We observe that in resource allocation problem the agent can learn more from one taken action. Our results show that the controller quickly converges to the optimal resource allocation policy and keeps the application performance within user-specified performance objective.

Chapter 6 presents an extension of approach described in previous chapter. One of the challenges of RL based approaches is dealing with state-action space complexity. The space size is determined by the number of parameters that describe resource capacity of the application VM. For multi-tier application we need to include into the state-space model parameters that define each tier capacity, which increase the space size. Large state-space RL model is hard to apply in practice. To address the issue we analyze the impact of individual VM resources on the application performance. The results of the analysis show that memory can be excluded from the application performance control knobs. It allows us to create separate RL models for RAM and CPU. By splitting models we significantly reduce the state-pace complexity.

In *Chapter 7* we investigate a batch application cluster sizing problem. MapReduce paradigm is *de-facto* standard for large data analytics, which has an open-source implementation called Hadoop. It offers a simple programming model for parallel computation with fault tolerance guarantees. Some of big public cloud providers adopted it and deliver Hadoop based cloud services for large data processing. To enable elasticity of MapReduce applications cloud providers split Hadoop cluster into data and compute parts. It allows easily shrink and expand the compute part of the cluster. However, to perform cost-efficient resource allocation the user needs manually determine optimal cluster size by running MapReduce job with different cluster sizes.

Chapter 7 presents a technique that finds an optimal cluster size during runtime from the first wave tasks of the job.

In *Chapter* 8 we summarize the content presented in previous chapters and provide our perspective view on future research directions.

BACKGROUND

2.1 INTRODUCTION

The number of virtualized datacenters and cloud hosting services is growing. Virtualization rapidly became popular, because it provides resource isolation, system manageability and reduced operational cost through resource consolidation. Some of data centers operate as a public cloud services, others are managed as a private enterprise clouds. Cloud computing offers so called 'pay-as-you-go' model. Cloud users can acquire and release resources on-demand. According the model, users pay for resources they use. Cloud market can be described as a stack consisting of three layers. Users and providers can interact on any of these layers.

Cloud infrastructure level, known as **Infrastructure-as-a-Service (IaaS)** defines models for monitoring, access and management of datacenter resources such as processing power, storage and networking services. IaaS eliminates the need to buy real hardware, users lease resources from a cloud provider. Examples of IaaS are Amazon EC2 [10], Google Cloud Engine [58], Microsoft Azure [101].

Platform-as-a-Service (PaaS) delivers environments and tools for management of cloud infrastructure. Users can build and deploy their applications, so they don't need to consider low level complexities such as configuring and setup of VMs. The examples of PaaS are Amazon Elastic Beanstalk, Google App Engine [57], Microsoft Azure Services [101].

Software-as-a-Service (SaaS) uses web to deliver application to the end user. SaaS moves the task of managing and deploying applications to the third party service. Among the examples of SaaS are Google Apps and storage solutions such as Dropbox.

In this thesis we mostly interested in IaaS level, where the process of virtual resources assignment takes a place. We consider a case, where cloud user wants to deploy an application. There is a need to answer the following questions. **How many** resources to allocate? **When** the user needs to trigger scaling? **What** performance levels can be achieved with allocated amount of resources? **Can** the user minimize the cost of running an application in the cloud?

Traditionally applications were provisioned statically. A company bought servers and ran an application on the private infrastructure. The resources available for the application did not change for a long time, until the increased workload reached the limit of the servers' capacity. As soon as it is happened, the company bought additional servers. Next time, when the limit was reached again, the cluster of servers was extended again. So it was a repetitive process. Cloud computing brought a new philosophy of resource provisioning-*elasticity*. In the nutshell the term *elasticity* means that users are able to stretch and contract resources dynamically based on resource demand from the applications they are running. Cloud providers charge users based on the amount of resources they use or reserve. But there is no guarantee with respect to the application performance that can be achieved with the assigned amount of resources.

It is common today that datacenters host a mix of interactive user-oriented and resource intensive batch applications [41, 143, 124, 119]. Interactive applications are latency sensitive and should not run under resource pressure. In contrast, batch applications are throughput-oriented and can tolerate resource pressure during runtime. Many applications running in the cloud environment have varying workload. Some variations are easy to predict. For example, promotion campaigns or load during working hours. But there are cases, where it is difficult to make an accurate prediction, such as unplanned load spikes caused by slash-dotting effects. Moreover, the application performance depends on the amount of allocated resources. For example, if we run an application on the server with 2 CPUs, then in ideal case we may expect that our application will be two times faster, rather than when it runs on a single CPU server. In other words there is a mapping between the resources assigned to the application and its performance. To achieve the desired performance the user has to change resource assignment every time when the workload changes. To perform accurate resource allocation it is required to take a number of steps: predict incoming workload, learn impact of workload on the application resource demand and find relationship between application resource assignment and its performance. This

```
IF <MetricName>{ComparisonOperator}<Threshold> FOR <T_history> //condition
THEN Change<ResourceName> by <AllocationUnit>                  //action
WAIT For <T_calm>                                              //calm period
```

Figure 1: Structure of scaling rule

is a challenging task, especially for the average user, who lacks expertise knowledge about the application.

To simplify resource scaling process cloud providers offer auto-scaling services. The services provide high level resource management interface, while users are responsible to define scaling policy. The policy is described in form of scaling rules. An example of the scaling rule structure presented on figure 1. A rule consists of action, threshold, metric and time delays, which determine when to trigger resource scaling process and how many resources to allocate. The user has to find right metric to track and the threshold value that triggers scaling action. In addition, the rule requires defining the amount of resources to allocate. One or two VMs? Which one is better? The auto-scaling service hides low level complexities from the user such as defining the host to run a VM, IP assignment process and etc. But leaves the user with a set of parameters to determine. It is hard to find a threshold value to achieve desired objective (e.g. minimize cost, maximize performance) without deep understanding of resource scaling process. Therefore there is a need to develop an auto-scaling system that is able to predict workload and perform scaling decisions to maintain desired performance with minimal amount of resources assigned to the application.

Below we outline the list of objectives that the auto-scaling system should fulfill:

- **Performance**. Cloud user expects certain quality of service delivery from a cloud provider hosting the application. To apply the requirements two parts (user and provider) sign contract called Service level agreement (SLA). SLA contains service level objectives (SLO). SLO describes performance that should be delivered by the cloud provider. For example, the application response time should not cross 1 second upper bound or a job should be finished within 10 minutes after the submission.

- **Cost**. Cloud providers charge users for the time the assigned resource is being used. Therefore to reduce the cost user either needs to minimize the resource usage time or maximize utilization of assigned resources. Otherwise, the user will overpay for underutilized resources.

- **Utilization**. The revenue of a cloud provider depends on the number of customers using the cloud infrastructure. Expanding datacenter is a costly process. Therefore in order to accept more users on a fixed size datacenter the cloud provider should focus on improving utilization of the datacenter computing resources. According to Delimitrou and Kozyrakis [41] modern dataceters utilization levels are around 10 – 20%. It means that there is a substantial room for the utilization improvement.

In the following sections we cover different aspects of resource scaling in the cloud. The background chapter is divided in to three parts. The first part presents resource scaling types. In the second part we overview cloud applications with respect to resource elasticity. The third part is dedicated to observation of cloud trends with respect to resource allocation. Finally, we summarize the topics discussed in the chapter.

2.2 SCALING TYPES

As it was mentioned above the key property of cloud computing is elasticity - ability to allocate and release resources on demand. The resource allocation in the cloud can be performed in

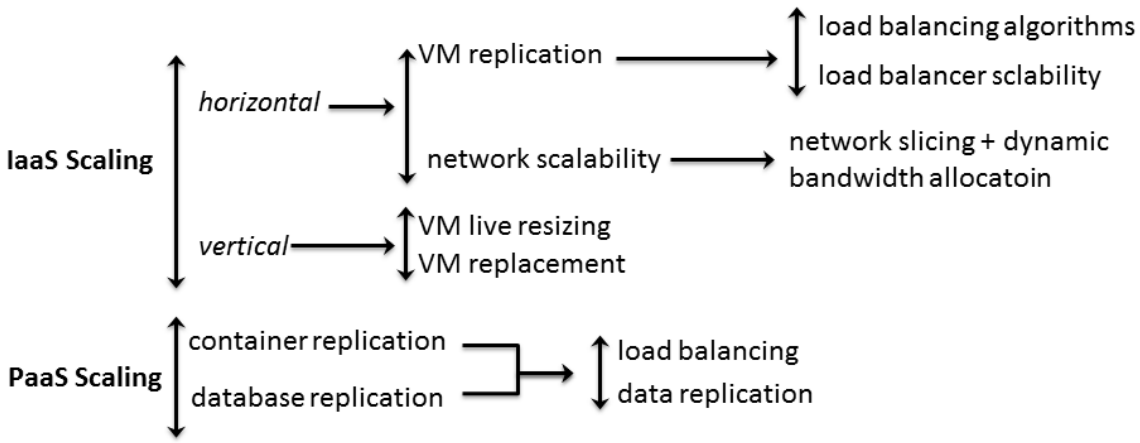


Figure 2: Summary of the Available Mechanisms for Holistic Application Scalability by Vaquero, Rodero-Merino, and Buyya [138]

different directions and on layers of the cloud stack. Figure 2 gives an overview of mechanisms available for scaling application in the cloud environment. The scaling actions can be triggered either on IaaS or PaaS levels. IaaS layer gives control over the cloud infrastructure in the form of virtual resources, while PaaS scales the application instances.

PaaS level hides management and scaling complexities from the user. The user does not need to install and configure a full stack of software. PaaS also simplifies the application scaling process. For example, database instances can be replicated to handle more queries. There is no need to worry about correct IP-addresses assignment or proper load-balancing. The new database instance will be added automatically to the database cluster. However, the process of replica management provides weak consistency [138]. Usually updates are not immediately done. There is an assumption that they will be 'eventually' done in the future. It poses some limitation of how transactions can be treated by developers.

Cloud is a multi-tenant environment, where applications from different users run together. Therefore there is a threat that users may interfere with each other on security or resource levels. To enable isolation some cloud providers such as OpenShift host application instances in the form of containers. The task of the containers is to provide necessary isolation. User applications are isolated with help of SELinux, CGroups, Linux quota and Linux namespaces. Number and size of containers can be easily scaled. However, the user is responsible to define the rules for scaling. The scaling rules are similar to the one presented in figure 1. Hence, the user faces the same challenges. PaaS level simplifies the process of the application development and resource scaling. But one major disadvantage of PaaS is lock-in problem. Users are locked to a certain platform and it is hard to move to another cloud provider. IaaS level gives flexibility in terms of changing cloud provider.

On the infrastructure level the resource management process is done via virtual machine. VM is a primary allocation unit in IaaS. The resource scaling can be implemented in two directions: **horizontally (scaling out/in)** or **vertically (scaling up/down)**. The first approach allows to add a new VM with an application server replica inside to distribute load among all running application servers. In contrast, vertical scaling allows resizing the application VM on-the-fly. For example, we can add more virtual CPUs (VCPU) to speed up our application. However, not every OS supports vertical scaling. Moreover, it is challenging task to collocate VMs that support vertical scaling. These are the reasons why only a few cloud provides support vertical scaling. But vertical scaling has a number of advantages over horizontal scaling that will be discussed further.

2.2.1 HORIZONTAL SCALING

Many applications on the cloud run in a distributed fashion. Therefore for these applications it is common when one of server instances joins or leaves the application cluster. For example, we can add more servers to the front-end tier of a web application, so it can serve more client requests. We can keep adding servers if the number of clients continues to increase. In ideal case we are not limited on horizontal scaling. Horizontal scaling allows expanding an application cluster even beyond single cloud provider [27]. Later when the load decreases, we can remove some servers if the remaining ones can provide acceptable performance.

From the first look horizontal scaling is simple. However, there is a number of issues to address. First of all we need a load-balancer that redirects client requests to the front-end servers. From resource provisioning perspective we have an overhead of allocating capacity to the load-balancer. Moreover, we need to run it continuously nevertheless the load is high or low.

Second, we need to perform IP-address management to inform the load balancer about a new application instance. Some of cloud providers such as Amazon offer higher customization level [26] that eliminates the problem. The load balancer is part of cloud provider's service and automatically routes traffic to a new VM instance. But it is not for free. The user pays for the time the load balancer runs and amount of data passing through it.

Third, horizontal scaling provides coarse-grained resource allocation. Most of cloud providers sell their resources in the form of 'T-shirt' sized VMs [55], such as *small*, *medium* and *large*. For example, Amazon EC2 *small* VM has 1 GB of RAM and 1 CPU, while *the medium* one has 4 GB and 2 CPU. Assume that our application requires only 2 GB and 1 CPU. Hence, we cannot run it on *the small* instance. We need to add 1 GB of RAM. Therefore we start *the medium* one. In this case the application will not utilize 2 remaining GB of RAM and 1 CPU. But the user will pay for all resources assigned to the VM. It is not what the user needs [4]. Users want to pay only for what they actually use.

Forth, adding a new VM is not an instant process. It has some overhead. Multiple authors [47, 8] evaluated horizontal scaling overhead. They found that it takes up to 1 minute to get a VM up and running. Is the value big or small? For the application that has to respond to real client the value is unacceptable. Probably a few people would like to sit in front of the screen and wait for 1 minute until the application they are working on, will respond to their request. It means that we have to take into account the resource *acquisition* time. The time it takes to start a new VM and get it ready to serve clients request. Large *acquisition* time requires triggering scaling process before we reach resource limit of running servers. Probably in example presented in 1 we cannot use CPU utilization threshold value equal to 100%. It should be lower. Alternatively, there are approaches that reduce horizontal scaling overhead with the help of VM cloning techniques [90, 23, 107] or VM live migration [31, 75, 146, 147]. VM cloning allows to start a copy of the running VM on another host. In case of live migration a VM is moved to the less overloaded host with the minimal service interruption. However, VM cloning and live migration techniques have an overhead in terms of CPU and network I/O that can affect performance of other VMs.

Some applications do not scale horizontally, because it causes a significant overhead. Relational databases do not support horizontal scaling out-of-the-box. To enable horizontal scaling one would need to partition the database by creating shards. It is a tremendous work to change data scheme of existing database of a company. NoSQL databases natively scale horizontally. But the scaling requires data transfer. For example, before adding a new instance of NoSQL database into production we need to run data re-balancing. The time is takes depends on the size of the data. The need to transfer data puts a limit on applicability of horizontal scaling for database applications. Alternatively, we can apply vertical scaling.

Cloud environment is a multi-tenant environment, where many users share virtual resources. Modern hyper-visors focus on providing guaranteed amount of CPU and RAM for VMs. How-

ever, they do not offer similar guarantees to the disk and network layer. As a result, it creates holes for possible resource stealing attacks [139] due to the lack of isolation for disk and network I/O. For many applications running in a distributed fashion the network plays substantial role in the application performance. Often it becomes a bottleneck. There is a number of attempts to deal with network saturation problems [5] and providing fair network share between user applications [87]. Baldine et al. [18] proposed the idea of Naas (Network as-a-Service) to offer capacity guarantees on the network layer. The approach can be implemented via distributed rate limiting [115], flow control [127] or network slicing techniques [102]. In NaaS the network bandwidth can be dynamically allocated based on the application demand. It is assumed that the users will pay for the actual bandwidth consumption. So far none of the well-known public providers offers performance guarantees for I/O level, either disk or network.

2.2.2 VERTICAL SCALING

Virtualization technology provides an alternative for horizontal scaling. Instead of adding more VMs, we can expand the size of a VM. Modern hyper-visors support memory, CPU and network scaling. We mainly focus on memory and CPU, because cloud providers do not offer dynamic network allocation. Further we describe the technologies used for memory and CPU scaling.

Memory scaling is performed with the help of memory ballooning [145]. The technique is used by hyper-visor to reclaim memory from a guest VM and share it with other VMs running on the same host. For example, a VM may allocate 4GB of RAM and use only half of it. In this case hyper-visor, can provision other VMs with the remaining 2 GB of RAM. The ballooning runs as follows. A balloon module is loaded into guest OS as a kernel service. To reclaim memory from the VM the hyper-visor instructs the module to 'inflate' by allocating physical pages within the VM. In the same way the balloon 'deflates' to de-allocate previously reclaimed pages. When the balloon 'inflates' it raises memory pressure in the guest OS. As a result it triggers memory management algorithms of the guest OS to free up memory. If the VM is low on memory, then some pages may be paged out to the virtual disk. Otherwise, if the guest OS has enough memory it will return free pages.

A VM CPU capacity can be changed in two different ways: by plugging virtual CPU (VCPU) and limiting physical CPU cycles available to the VM. CPU plugging is a technique that allows adding or removing virtual CPU to/from a VM during runtime. Historically CPU hot-plug feature was used to isolate failing CPUs and later on a number of other use cases appeared [54]. Such as clearing work from CPU, improving energy efficiency and resizing guest OS running in virtual environments. Unfortunately, not every OS supports the technique. Linux supports CPU-hotplug starting from version 2.6.5. If CPU-hotplug is supported, then it takes about 150 ms to bring CPU offline and 220 ms to bring it online [110]. It is two orders of magnitude less than adding a new VM in public cloud. If CPU-hotplug is not supported, then one can use a hyper-visor's virtual CPU limit capabilities.

CPU limit does not require the change of the kernel. It can be applied to any OS. Xen [19] and other hyper-visors offer CPU scaling feature. To describe how the CPU scaling works we use Xen credit scheduler as an example, since Xen is one of widely applied hyper-visors [29]. Each VM VCPU is given a credit, which represents share of physical CPU. The scheduler monitors CPU usage every 10 ms and removes the credits from currently running VCPU. It switches to another VCPU if the current one has no credits left. The credits are given to a VCPU every 30 ms. Hence, if CPU intensive VM runs out of its credits, then it has to wait for 30 ms. The scheduler supports work-conserving(*wc*) and non-work-conserving (*nwc*) modes. In *wc* mode Xen assigns a *weight* to each VM. For example, in case of resource contention a VM with weight of 512 will get twice as much CPU as a VM with the weight of 256. However, if one of the VMs is blocked, then the second one can utilize entire CPU. In *wc* mode idle CPU cycles distributed among running VMs. In *nwc* mode each VMs gets *cap* value. If *cap* is zero, then

VM does not receive any extra CPU cycles. Cap value above zero expresses amount of CPU VM gets in *nwc* mode. Public cloud provides such as Amazon use *cap* value to fix the power of VM VCPU [11]. For example, if a host processor runs at 2GHz and a VM *cap* value is 50% then the speed of VCPU is 1GHz. *nwc* mode reduces CPU sharing efficiency, but improves isolation. CPU limit as well as CPU plugging has sub-second range allocation overhead [126]. Existing auto-scaling systems use either CPU plugging or CPU capping techniques to change VM computational capacity.

In comparison to horizontal scaling, vertical scaling allows to assign virtual resources in a fine-granular manner. For example, we can choose between adding 512MB RAM or 1 GB. Figure 3 shows an example how vertical scaling helps to deal with increasing workload. In [39] authors compared *ElasticVM* and *StaticVM* that run Apache web server. *ElasticVM* was allowed get 30% more CPU power than *StaticVM*. Authors used step function to increase traffic rate to the VM. Up to 730 seconds both VMs provide same performance for increasing workload. However, after 730 seconds *StaticVM* reaches its resource limit, while *ElasticVM* continues to expand. On the right hand side graph we see that fully utilized *StaticVM* dramatically increases response time, while *ElasticVM* keeps response time below SLO bound (20ms). The presented example shows that vertical scaling sustains application performance when the load increases. As it was mentioned above the overhead of horizontal scaling is about 1 minute. Vertical scaling can provide necessary time buffer to freshly started VM. In our example we could start horizontal scaling at 730 seconds and continue to scale vertically. For the time *ElasticVM* reaches it resource limit on the host the new VM on another host will be ready to serve incoming requests. Combining horizontal and vertical scaling allows scaling beyond capacity of the host with minimum negative impact on the application performance.

However, vertical scaling is not always possible or it creates challenges when applied in cloud environment. Many applications are not designed with resource elasticity in mind. For example, some applications have static resource limits parameters. Especially it is true for memory. Java applications have the heap size parameter that limits memory available to JVM. Hence, extending memory capacity of a VM will not affect the Java application running inside the VM. We need to update the heap size limit too. Moreover, scaling down below the heap size may lead to out-of-memory events (OOMs) [143]. Therefore the application should be aware of a resource elasticity. It raises challenge on developing the applications that adapt for expanding and contraction of virtual resources. In [68] authors manually instructed Apache web server to flush used RAM to scale memory down. Salomie et al. [122] addressed the problem of dynamic memory allocation for the application with statically configured memory. They extended ballooning to a database engine and java runtime, so that memory can be efficiently distributed between virtualized instances. [94] adapted java process heap size based on the execution of previous tasks of MapReduce job. Another challenge is the complexity of scheduling VMs with vertical scaling capabilities. The size of the VM can change at any time, so it becomes difficult to collocate it with other VMs. However, there are approaches [153] that address collocation of VMs with vertical elasticity. In chapter 4 we present an approach to collocate VMs with vertical scaling capabilities.

To summarize, the resource scaling on IaaS level can be performed horizontally or vertically. Horizontal scaling uses a VM as resource allocation unit and allows boundlessly scale the application out. However, it comes with substantial overhead for client interactive applications. Hence, the scaling process should be started long enough before the incoming workload reaches the capacity limit of the VM. Some application types such as databases do not support scaling out or they require significant time and network I/O to add new instance (we need to transfer data to the replica). Vertical scaling has lower management complexity. There is no need to assign IP-address or requests redirection. Vertical scaling allows fine-granular tuning of individual VM resources, so the application can be provisioned with only resource it needs. But the scaling horizon is limited by capacity of the host. The process of scaling up/down has an

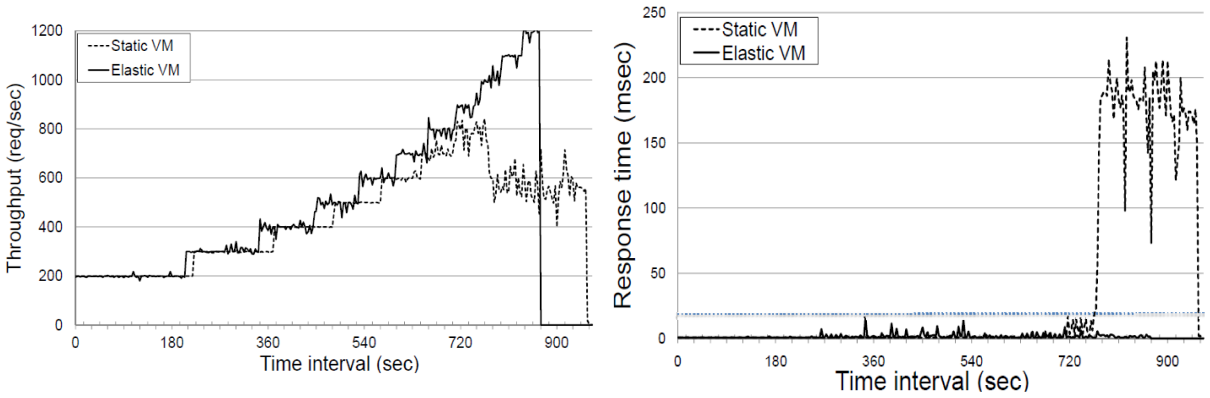


Figure 3: StaticVM vs ElasticVM: Throughput and response time comparison reported by Dawoud, Takouna, and Meinel [39]

overhead in sub-second ranges, which makes it attractive to deal with sudden workload spikes. However, to take full advantage of vertical scaling and achieve the best application performance the internal parameters of the application should be adapted. The parameters adaptation is out of the scope this work.

2.3 WORKLOADS

A wide range of applications runs on cloud infrastructure. Web applications, batch applications, video streaming services are the examples of common cloud workloads. Each application has its own performance constraints and preferable scaling horizon. Some applications are critical to resource shortage, because it greatly degrades their performance. Others can tolerate the shortage and make a progress even under high resource pressure. Moreover, not every application can be easily scaled vertically and horizontally. Awareness of the application type, its performance requirements and resource scaling limits allows the auto-scaling system to make better scaling decisions. Therefore in this section we present classification of applications in the cloud.

There are number of works [41, 143, 124, 119] that analyzed modern datacenters workloads. The datacenters workloads fall into one of the two classes: interactive and batch applications. The first class of the applications often called as latency-sensitive [143] or user-facing interactive applications [41]. Interactive applications usually serve real clients over *http* protocol. The examples are online shops, Google services such Gmail, Google Docs or web search engines. Since interactive applications communicate with real users, it is important to provide human acceptable performance. Usually response time for these applications should be in sub-second range. Violating the requirement can lead to high revenue loss. Amazon reports that every 100 ms of latency can affect the sales by 1%. Another observation from Google [100] states that increase of page load time from 400 milliseconds to 900 milliseconds results in 25% drop of the traffic. It means that clients get disappointed and go to another web site.

Interactive applications run indefinitely. The workload of the application has different patterns: daily, weekly, seasonal, bursty. The first three patterns can be easily predicted with high accuracy. We know that clients are more active during daylight time. So it is expected that the application has to serve more requests when people at work, rather than at night, when most of the people sleep. There is also longer period pattern such as difference between working week and weekends. Some e-commerce applications are affected by higher load during special seasons. For example, people tend to buy more during Christmas time. All of these patterns are easy to predict, so one can plan resource provisioning in advance. However, bursty load prediction is a challenging task. It characterized by high fluctuation and short duration. The reasons for such load could be different. For example, the release of a new product may attract more

than the expected number of internet users. Or if the news published on a highly popular website refer to some less popular one. Then a high fraction of the clients get redirected from the first website and overload the second website. Such situation is called "slashdotting" effect. To deal with bursty load one could always keep some headroom of resources. However, it is obviously leads to resource wastage, since load spikes are rare. As an alternative one could use reactive scaling. There is no need to allocate resources upfront. But it requires low overhead resource allocation mechanism, to minimize impact on the application performance. Vertical scaling is the best candidate for reactive provisioning. It has a sub-second resource allocation overhead.

Most of interactive web applications have tiered architecture: front-end tier (web server), application tier (business logic) and back-end tier (database). The incoming requests go from tier to tier and come back to the clients with results. Each tier of the application can be scaled separately. But it does not mean that one tier can be scaled in isolation of other tiers. Scaling mechanism should be aware of cluster wide correlation and possible shift of bottlenecks. Cluster wide correlation happens when one of the tiers has certain resource shortage that affects resource usage of another tier. For example, CPU saturation of database tier raises memory utilization of the front-end tier. It happens because the database cannot serve fast enough incoming requests. Hence, it cannot accept connections from the web server. As a result the web server has to enqueue incoming client requests. The bigger queue needs more memory. Horizontal scaling is one of commonly used resource allocation mechanisms for interactive applications. However, horizontal scaling of back-end tier is challenging. The back-end tier is a state-full tier. We cannot just start one additional instance. We need to transfer the data from a running instance to the replica. Moreover, it is desirable to shrink resources when the load decreased. Shutting down one of database instances requires special treatment, otherwise the data will be lost. For this reasons most of existing auto-scaling systems [63, 82] leave it over-provisioned and don't scale it. One can argue that there are number of distributed key-value stores which can be easily scaled out and scaled in. Unfortunately, most of e-commerce systems use traditional relation databases, which don't scale horizontally. To enable horizontal scaling one would need to build sharding logic into relational database, which introduces complexity and limits ability to use the relational features of the database. A good alternative is vertical scaling. It eliminates the need for data replication. We can easily grow and contract a VM running database instance.

Batch applications consist of resource intensive jobs, which can start and stop at any point of time. These applications usually perform data-analytics tasks to solve scientific problems, crawl internet and create different types of reports. Large body of data analytic platforms use MapReduce computational model [40]. Hadoop is one of popular implementations of MapReduce paradigm. A number of companies use it in a production [16]. SLO objectives of these types of applications are described in the form of job execution deadline. Batch job consists of a number of tasks that can run simultaneously. Therefore resource allocation process of batch applications is performed in the form of task scheduling. The scheduler decides how many tasks to launch in parallel to meet the deadline. It means that the scheduler performs horizontal scaling. The task usually runs in a container, which limits resources such CPU and RAM available to the task. Batch applications also support vertical scaling. To improve task runtime Li et al. [94] adapted the container size based on previous tasks execution. Batch jobs are tolerant to performance slowdown and could run under high resource pressure. Hence, in contrast to interactive applications batch applications do not require immediate reaction to resource under-provisioning. Moreover, in case of resource shortage on the host one could apply admission control via task termination and relaunch the terminated task later [30, 32]. These jobs are designed with fault tolerance in mind and termination of one the tasks does not kill the application.

As we stated earlier batch jobs are resource intensive and can put pressure on any of the

resources: memory, I/O, CPU. Therefore, a lot of research has been done to deal with various resource bottlenecks. Over the past decade disk locality was the main focus in resource usage optimization [156, 14] for batch applications. The motivation for disk locality is based on two facts [13]. First, disk bandwidth exceeds network performance. Second, disk I/O has substantial contribution to overall task execution time. Nowadays the focus shifts from disk locality to network. It was observed that network reads performance is comparable to the disk reads [72]. Batch jobs can create a huge traffic across the cloud. It can affect performance of other applications sharing cloud infrastructure with them. Hence, the auto-scaling system should consider, not only resources resided on a host, but also pressure on the networking infrastructure [6, 128, 36].

One of the main goals of cloud provider is minimize energy costs related with cloud infrastructure. For cloud provider it is difficult to expand infrastructure, because power source with desired capacity may not be available nearby. Hence, cloud provider needs to push datacenter to higher utilization levels. Current state of utilization levels far away from ideal. It is about 10 – 20% for the industry [41, 143]. One of the reasons is that datacenter owners sacrifice utilization for latency sensitive applications. These applications run on dedicated servers in isolation from batch applications. In isolation latency sensitive applications achieve high performance even during peak demands and sudden load spikes. The modern trend is to collocate both application types and raise utilization of the datacenter [32, 143, 41]. The desired performance of latency intensive application is achieved with help of prioritization [41, 143] and vertical scaling [153].

To summarize, cloud runs a wide range of workloads. Cloud applications can be classified in two groups: batch and interactive applications. The first group contains resource intensive jobs that partitioned into tasks. The jobs can tolerate significant wait time. The scaling of the applications implemented in the form of scheduling, where the scheduler based on available cluster capacity decides how many tasks of each job to run. In contrast, the second group consists of applications that are highly sensitive to performance slowdown, because they serve real clients. Even a short delay can cause high revenue loss. Interactive applications consist of tiers. Hence, the auto-scaling system should orchestrate resource scaling across all tiers to avoid shift of resource bottlenecks.

2.4 CLOUD TRENDS

Most of IaaS cloud providers sell virtual resources in the form of fixed bundles such as VMs. The bundles have predefined amount of CPU power, memory and storage size. Providers charge users for the time the resource bundle being used. The users usually charged for VM usage in a hour granularity. Agmon Ben-Yehuda et al. [4] analyzed development of the cloud market and observe number of trends that drive cloud market toward fine-granular resource allocation, seconds range billing cycles, proper resource pricing and service level differentiation.

Billing periods Years before cloud computing has emerged, the time for which physical server was used counted in years. The appearance of web hosting changed the situation. Clients were able to rent servers on a monthly basis. In 2006 Amazon introduced elastic compute cloud (EC2), which dropped rental granularity from months to hours. So the cloud users were allowed to rent servers for hours and shutdown unused ones. Hence, the users paid only for resources being utilized.

The renting of VMs for shorter periods is driven by economic incentives, which push users to perform optimizations in order to reduce resource wastage. For example, if an instance runs for half an hour, then user has to pay for full hour. It means that the user overpays for half an hour. However, if half second runtime of a VM was billed as full second, then the user overpays only half a second. Smaller billing cycles reduce user's overpay costs.

The trend of shrinking billing cycles is already in the cloud market. In 2009 Amazon announced *spot-instances*, which price changes every 5 minutes [2]. Later on in 2010 a new cloud provider CloudSigma [33] announced 5 minute billing cycles. Then in 2012 GridSpot [59] and ProfitBricks [113] appeared in the market and offered 3 and 1 minute billing cycles, respectively. In 2013 Google Cloud Engine started to charge by 1 minute. The trend of shrinking resource billing cycles runs parallel to telephony billing cycles. In the past telephony billing cycles shrank to seconds. Therefore, there is an expectation that cloud providers continue to move towards second billing cycles.

Resource granularity Selling fixed bundles is common for the most of IaaS providers. Cloud providers have different names for the bundles. For example, Google Compute cloud calls them 'machines type', Amazon and RackSpace call 'instance type', GoGrid calls 'server sizes'. Cloud providers use fixed bundles to keep a connection between virtual resources and real hardware. So the user can find virtual machine that is equal to a server running in private infrastructure. From 2012 the model of fixed bundles begins to change. In 2012 Amazon announced Elastic Block Storage service that allows users to modify I/O resources for already running instances. Freshly started cloud providers such as CloudSigma(2010), ProfitBricks(2012) and GridSpot(2012) announced flexible bundles model. Users can specify the amount of resources they need. It is similar to buying server in computer shop, where the customer decides how much RAM, CPU and hard disks put into a server.

Users are not motivated to buy fixed size VMs. The workload of many applications is not constant. It changes over the time. Hence, the VMs are not fully utilized every moment of the time. Moreover, the workload may change from CPU intensive to memory intensive, which would require changing type of the VM. Hence, the model of flexible bundles enables following the workload dynamics and optimizing resource usage. The dynamic fine-granular resource allocation from one side leaves space for optimization; from another side it increases the complexity of resource management. The user would need automating tools that are able to allocate resources according to the price of the resources, the application workload and the user's objectives in terms of cost and performance goals.

Resource pricing Multiple VMs running on the same host with a help of virtulization technique can successfully share CPU and memory resources. However, the sharing of network and disk I/O is more challenging. It was observed [17, 139] that performance of VMs can greatly vary due to interference and bottlenecks created by collocated VMs. It means that there is a difference between what user rents and what cloud provider delivers. To solve the issue researchers proposed to sell the performance instead of resources [109, 136, 106]. The approach is easy to implement on the levels such SaaS and PaaS, where application performance can be well defined and cloud provider has full visibility of the user's application. However, on IaaS level cloud provider and the user are separated entities. The provider does not know what kind of application the user runs and what performance levels are desired. Moreover, the user could lie to the provider about the application performance to get more resources [3]. In order to assure the application performance cloud provider could offer resources in the form of guaranteed time of resources. For example, in 2014 Amazon launched burstable T2 instances and general purpose SSD (GP2). Each VM has certain guaranteed CPU minutes per hour. It means that VM gets guaranteed baseline CPU speed for these minutes. Moreover, according to the model the user can earn credits when a VM is idling (1 credit - 1 guaranteed CPU time). Later when the load burst occurs the VM gets guaranteed CPU cycles. Such model allows users to get VMs for lower price and make sure, that it will provide the best performance during high load spikes.

Service levels Instant resource demand from the users can exceed cloud provider's datacenter capacity. However, the users have different performance objectives. Some users have strict performance requirements, others are more flexible. Therefore, to be able to accept resource requests from all users cloud provider later during runtme can preempt or slowdown applications of flexible users for the sake of strict ones. In 2009 Amazon introduced *Reserved*

instances and *Spot instances*. *Reserved instances* are high priority level VMs. *Spot instances* are low-priority level VMs that usually have significantly lower price in comparison to *reserved instances*. By prioritizing VM instances a provider can offer elasticity and availability to high priority users for the cost of degrading low priority users. However, cloud provider can also offer spare resources to low-priority users if high-priority users don't utilize acquired capacity. Both sides of the cloud market can benefit from the service differentiation. Cloud providers have opportunity to better utilize the infrastructure by offering residual resources in the form of low-priority VMs. Cloud users can choose priority level that reflects their budget and the application performance constrains. For example, users with smaller budget can enter cloud by using low priority VM.

We gave an overview of trends in the cloud market. There is evidence that cloud providers' resource rental model moves from coarse-grained toward fine-granular resource allocation. Moreover, cloud providers billing periods in the near future are going drop to the seconds range. Till now the shift from hours to minutes was already observed. **Smaller billing periods together with fine-granular resource allocation increase the complexity of resource allocation in comparison to fixed size VM assignment.** The new model would require development of auto-scaling systems that are able to find optimal decision with respect to workload dynamics and flexible resource bundles model offered by the cloud environment. The users of the cloud are diverse in terms of budget and the application performance constrains. Therefore cloud providers would benefit if they offer resources under different service levels. Such approach opens access to the cloud for the users with smaller budget and offer guaranteed performance for high priority users. In this circumstances the role of vertical scaling increases. Instead of migrating VM during peak load, it allows with low overhead upscale one user VM by taking resources from neighboring VM of another user.

2.5 SUMMARY

We started with an overview of cloud market and presented the problem of dynamic resource allocation. Then we looked into resource scaling types: described horizontal and vertical scaling. We discussed overheads and applicability of each of the types. We also pointed out that to achieve the best performance out of allocated amount of resources one needs to adapt the application parameters too.

Later we analyzed applications running in the cloud environment. Cloud applications can be classified in two types: interactive and batch applications. Interactive applications are latency sensitive and require either some resource over-provisioning or low overhead reactive scaling to keep the latency low. Batch applications usually run resource intensive workloads and can tolerate performance slowdown. Resource allocation for batch applications is usually defined as a scheduling. The goal of the scheduling is to meet certain job execution deadline. The workload of the both application types can change during runtime. It can vary during day, week, etc. Some of the workload patterns are hard to predict, such as load spikes caused by 'slashdotting' effects. In this case we can apply vertical scaling that has low resource allocation overhead. Moreover, the workloads can change from being CPU-intensive to memory-intensive.

Finally we provide survey of cloud market trends. The major outcome is that resource rental model moves from fixed bundles to flexible bundles, where cloud user can dynamically change individual resource assignment. However, fine granular resource allocation increases the complexity of capacity management process. Hence, there is a need for auto-scaling systems that can provide optimal scaling decision from the large number of possible allocation schemes. Another trend is service differentiation. By offering different service levels cloud providers can better utilize own infrastructure and lower barrier of entry for small companies.

AUTO-SCALING SYSTEM

3.1 INTRODUCTION

The goal of auto-scaling system is allocate resources to effectively handle dynamic workload changes, while providing guaranteed application performance with respect to SLO. Resource auto-scaling is a complex process that requires taking a set of steps. First, one needs to monitor workload, application status and resource usage. Second, to make a decision about resource provisioning it is necessary to analyze monitored data. Third, as soon as decision about resource allocation has been made, we need to plan how many resources to allocate. To perform optimal resource scaling decisions the accurate model of relationships between the application performance, input workload and the amount of allocated resources should be identified. The questions are *when* and *how* to identify the relationships. The final step of auto-scaling process is allocating the estimated amount of resources. In section 3.2 we describe auto-scaling process phases in details. Then in section 3.3 we present system identification approaches and discuss pros and cons of each the approaches. The relationships can be described using a wide range of techniques. Among the literature we review the techniques applied in auto-scaling systems fall into following categories: threshold based techniques, reinforcement learning, techniques that use control and queuing theories, and techniques that based on time series analysis. We present an overview of these techniques and related work in section 3.4.

3.2 AUTO-SCALING PROCESS

The process of auto-scaling has a set of phases and can be described as MAPE loop [49, 77], where M stands for monitoring, A - analyzing, P - planning and E - execution. The collecting of information about resource utilization, input workload and the application status is a part of monitoring phase. Analyzing phase uses monitoring data to determine current state of the application and estimate the need to perform scaling action. In planning phase auto-scaling system decides what action to take. The system needs to find tradeoff between cost of resources and satisfying user-defined SLO. Finally, in the execution phase via API call of underlying platform the actual resource allocation action is performed. The auto-scaling system has four components (monitor, analyzer, planner, executor) that involved in each of the phases.

The **monitoring** is essential part of auto-scaling system. It provides measurements about resource consumption, the application health status and its performance with regard to objectives defined in SLA. The list of monitored metrics consists of CPU, memory, network, disk utilization, response time, throughput,request rate job progress and etc. Some works [126, 108] use CPU as the application performance indicator. However, better performance control is achieved if direct metrics such as response time included into monitored metrics set [85]. The quality of scaling decision also depends on the delivery of up-to-date monitoring data. To monitor applications researchers apply sampling intervals from seconds to minutes. Emeakaroha et al. [49] evaluated the impact of monitoring interval size on the web application SLA violation rate. They use 0.15$ as a measurement cost and 0.30$ as a cost of missing violation. 5 seconds measurement interval was set as baseline that detects all SLA violations. Figure 1 presents results of the evaluation. The graph shows that the cost of missing SLA violation significantly increases if the monitored data updates delayed. Hence, monitoring component should be in sync with the application it monitors. Most of cloud providers offer monitoring services on IaaS level [26, 1]. However, provider services are not always exposing all necessary information. It is common that the user needs implement own monitoring agents to collect the application specific data.

The **analyzer** processes monitoring data to analyze application status and resources utilization. It triggers planning phase if the SLA violations or significant changes in workload characteristics are detected or predicted. Auto-scaling system is considered is *reactive* if it responds only on the current state of the application. It means that the planner will scale assigned re-

Figure 1: Impact of monitoring interval

sources when SLA violation is already happened. Resource allocation is not an instant process. It has some overhead [52]. For example, adding one VM requires up to 1 minute [47]. Hence, it is important to trigger the planner with some anticipation. The system that anticipates future resource demand is called *predictive*. Predictive scaling provisions the application in advance to deal with fluctuating workloads. Most of works presented in 3.4 use time series technique to analyze the monitoring data.

The **executor** performs actual resource allocation: add/remove VMs, resize capacity of running VM or even migrate VMs. The allocation actions performed over API offered by cloud provider. If the application deployment runs on private cloud, then API of cloud management framework is used. For example, CloudSigma provider offers REST interface to dynamically resize VM. As we stated earlier in section 2.2 actual resource allocation process has some overhead. Therefore scaling overhead should be part of the planner's resource assignment algorithm to prevent the application performance degradation during resource assignment.

The **planner** component is a core of auto-scaling system. It contains a system model which describes relationship between the application performance and its resource demand. The model can be fixed during design time or the planner can adapt it during runtime. As soon as the information about current or predicted state of the system is known, the planner has to make a decision about resource assignment. For example, VM can be removed from application cluster if the observed application utilization is low. However, the assigned capacity should guarantee desired application performance. As we stated earlier in section 2.3 the workload of many applications constantly changes. Therefore the planner should also include input workload into the system model. In section 3.4 we will cover auto-scaling techniques and algorithms that compose the core of the planner.

3.3 SYSTEM IDENTIFICATION

The resource scaling policy is applicable only if the model of controlled application is well investigated. Therefore to develop resource scaling system, first we have to understand the quantitative relationships between the application performance and its resource usage. To build a model (or identify system) we need a profiling environment. The environment can be fully isolated from the application running in production mode(offline modeling), build on top of production environment (sand-boxing) or the application running in production mode can be profiled during runtime (online modeling). Each of the approaches has strong and weak sides. One can choose the approach based on optimization goals.

Many auto-scaling systems [43, 108, 67] use fixed models that obtained **offline** and do not change during runtime. To discover the model authors of the systems take empirical approach

and run set of tests against the application that will be later put into production. The goal of the tests is find and tune essential application and scaling policy parameters. The approach provides high accuracy, since it easy to repeat the experiments, compare results with previous runs and tune the model parameters. However, any changes of the application such as updates or unseen workloads would require going offline again and running tuning experiments again. Moreover, in some cases it is difficult replay real workload. There is a need for approach that builds the system model during runtime.

Sand-boxing [159, 140, 41, 143] eliminates the problem of offline system modeling. The idea is to build a small clone of the application production environment and redirect live workload to the sand-box. The overhead of profiling the application and adapting scaling policy can be greatly improved. Delimitrou and Kozyrakis [41] claim that it requires about 5 minutes to obtain a new profile. Experiments with the sand-box do not affect performance of the application in production mode, since the 'sand-box' isolated from the application running in production mode. However, the approach has some overhead in terms of infrastructure for the 'sand-box' and its management. First, the complexity of building sand-boxing environment is high. One has to provide application specific implementation to redirect live workload data. Second, it requires dedicated hardware for testing scaling policy. Applications hosted on the cloud can be very different by nature. Hence, it is hard to build generic 'sand-box' for all applications. It can be only application specific, which also stated by authors in [41] .

In the **online** system identification approach the application runs on the cloud without maintenance delays and the scaling policy immediately adapts to the application resource demand. Machine learning techniques [118] and statistical analysis [22] among the tools applied in for the approach. The strong side of the approach is that it does not require upfront knowledge about the system model. The resource scaling policy is obtained by observing the reactions of the application on resource assignment actions. However, it requires careful change of resource entitlements, because the actions applied to the life system. The time to obtain initial policy can be significant, because one needs to collect enough observations to train the model. But there are approaches that address the issue [154, 20].

To summarize, we can use different system identification approaches. Testing allocation policy offline gives the freedom of experiments. However, we cannot catch all real world complexities. Sand-boxing brings the testing environment closer to the real cloud environment. So we can evaluate scaling policy on a live workload data. But the approach has overhead of building the sand-box. Moreover, we would need to adapt the sand-box for every application we want to dynamically provision. Evaluating scaling policy of the application running in production mode eliminates the drawbacks of two previous approaches. However, it adds the challenges to the system identification process. Because adapting scaling policy online could impact performance of the application. The process of system identification should be less visible to the application client. Otherwise, bad experience with application would motivate clients to move to other services [100].

3.4 AUTO-SCALING TECHNIQUES

The goal of auto-scaling technique is determine the relationships between application performance and assigned capacity. The techniques that used to model the relationship can be divided into five categories: threshold based, reinforcement learning, control theory, queuing theory and time series. Threshold based, control theory and queuing theory are offline system identification approaches. The model is obtained analytically or empirically over a set of offline experiments. And the model does not change lifetime of the application. Some control theoretical approaches can perform online system adaptation. But, the initial model is obtained offline. Auto-scaling systems that exploit reinforcement learning and time series can be used for online system identification. However, both techniques require significant time to obtain initial policy.

```
1: if x > thUp then
2:     add y
3:     wait for inUp min.
4: end if
```

```
5: if x < thDown then
6:     remove y
7:     wait for inDown min.
8: end if
```

Algorithm 1: Scaling up/out and scaling down/in rules

But there are works [56, 126, 116, 154, 20, 152] that address the issue. In this chapter we describe each technique and present related works.

3.4.1 THRESHOLD BASED SCALING

Many cloud providers offer auto-scaling services that exploit threshold based scaling. The popularity of the technique is explained by its simplicity. It is easy to understand for average cloud user. However, to design a good scaling policy the user is required to have deep understanding of the application, its workloads and experience with provider's infrastructure.

Threshold based techniques use rules to describe scaling policy. The rules define amount resources to allocate/de-allocate such as number of VMs, CPU or RAM. The rules can be divided in two sets: scaling up/out or scaling down/in. Algorithm 1 shows examples of the rules presented in two columns. The left side column describes allocation (scale up/out) rule and the right side column shows de-allocation (scale down/in) rule.

The rules consist of two parts: condition and action. The lines *1* and *5* are conditions of the rule. Usually the condition contains the metric x and its threshold $thUp$. The condition can be also complex and contain more metrics $x_1, x_2, .., x_n$ and thresholds $thUp_1, thUp_2, .., thUp_n$. It is common to use CPU utilization [26, 85, 63, 64] and response time [85, 74] in conditions. The second part is action. In our example actions presented in lines *2* and *6*. For horizontal scaling the action is defined as a number of VMs to add or remove. In case of vertical scaling individual resources such as CPU, RAM is described in the action. After the action is triggered the controlled application requires some time to reach steady state, therefore the rules have 'grace' period between scaling actions: $inUp$ and $inDown$. This is the time, when no action is executed even if the conditions are met.

Usually threshold based rules contain only two rules per metric: one for extension and one for compaction. Hasan et al. [64] extended the rules. In addition to the upper and lower thresholds they added intermediate upper $ThrbU$ and the lower $ThroL$ thresholds. The former is slightly below the upper threshold and the latter slightly above the lower threshold. The authors also added two duration parameters. The main goal of extending the rule is to catch the trend of measured metric. However, the question about how to determine 'good' threshold values remains open.

The quality of resource scaling algorithm greatly depends on the metric used to trigger scaling action. In [26, 85] authors compared utilization-based and latency-based rules. For utilization based rules they applied CPU utilization, while for the second approach average response time was used to trigger scaling actions. The results from both rules show that applying direct application performance metrics such response time allows to save resources in comparison to utilization based rules. The application running under resource pressure still can achieve desired performance goal.

Most of threshold based auto-scaling systems operate in a reactive way by executing scaling action after the threshold is met. Casalicchio and Silvestri [26] added one step ahead arrival rate prediction mechanism to the threshold based scaling. The evaluation results show that the

prediction allows significantly improve response time of the web application. Moreover, it gives cost savings about 15% in comparison to only threshold based policy.

While most of cloud providers offer simple rules, RightScale added voting feature to threshold based approach [125]. Each VM within a cluster of VMs can vote for grow or shrink action based on its utilization levels and defined threshold. The threshold is equal across all VMs. The scaling in the system is triggered as soon as the majority of VMs vote for the scaling action. The value that defines the majority can be tuned as well. RightScale also defines calm period between the actions, which prevents the scaling algorithm from instantly booting new VMs. The proposed algorithm is just an implementation of democratic voting. The task of defining thresholds still remains on user's shoulders.

The threshold based scaling was adopted to provide not only horizontal scaling, but also vertical scaling. Han et al. [63] implemented auto-scaling system to perform response time control of three-tier web application. The system scales VM resources such as number of virtual CPUs, RAM and I/O. However, scaling granularity is not very high. RAM is changed by 1GB, disk bandwidth by 10 MB/sec. To trigger scaling actions the authors use empirically defined thresholds and no prediction technique is used. The system is tested against static workload that does not reflect real workload dynamics.

One of the difficult parts of defining the rules is determine the upper *thUp* and the lower *thDown* thresholds for a metric. The higher *thUp* allows to delay resource assignment and hence, rent fewer resources and save money. However, it could potentially lead to performance degradation of the scaled application [130]. With lower *thUp* we can achieve good performance, but then we have higher level of over-provisioning. A large body of research papers applies predefined thresholds for rule-based scaling policy. Suleiman et al. [130] analyze the tradeoff between different scaling thresholds for horizontal scaling. In particular they experiment with upper threshold for CPU utilization and response time. The other parameters such as grace period and provisioning unit are fixed. The evaluation is done on Amazon EC2 infrastructure. Authors observe that higher thresholds reduce server usage cost in comparison to lower thresholds. Moreover, they found substantial difference in acquisition time between *small* and *medium* VMs. The first one takes about 5 minutes to start and the second only 2 minutes. It means that the thresholds should be tuned not only per application basis, but also per VM type.

VMs instances of an application can be terminated as soon as lower threshold bound *thDown* is met. However, applying the scaling down rule without taking into account a cloud provider billing model can lead to higher costs and worse application performance under highly fluctuated workload. Casalicchio and Silvestri [26] and Kupferman et al. [89] address the problem of saving VM usage costs based on how cloud providers charge for virtual resources. Many cloud providers have one hour billing cycle. Hence, it is better to keep running a VM before billing hour is over, even if the overall utilization is low. The approach is beneficial in case of highly fluctuating workload, because there is always some amount under-utilized VM capacity.

Interactive applications are the main focus of rules-based auto-scaling systems. The most of proposed systems address scaling of single tier interactive applications. In [45, 26, 130, 85] authors target application tier and only a few of the works address scaling of more than one tier[63, 64]. Scaling more than one tier requires dealing with cluster wide correlation [116] to make sure that bottleneck resource does shift from one tier to another. For example, CPU saturation of the database tier leads to higher memory usage of the web server. In [63] authors do not address cluster wide correlation. The threshold based approach was also applied to storage tier. Lim, Babu, and Chase [95] developed automated controller for elastic storage based on Hadoop Distributed File System(HDFS). The controller uses predefined thresholds to trigger horizontal scaling actions.

In summary, the rule based scaling is the first step towards auto-scaling systems. It provides easy to understand rule semantics. However, the setting up the rule parameters is a challenging

1: $Q \longleftarrow Q_0$ ▷ initialization e.g. $Q_0 = 0$
2: s ⟵ SelectState()
3: **while** not terminate **do**
4: a ⟵ SelectAction(π, s) ▷ policy π from Q e.g. ε -greedy
5: r′ ⟵ Reward(s,a)
6: s′ ⟵ NextState(s,a)
7: $Q(s, a) \longleftarrow Q(s, a) + \alpha[r' + \gamma * argmax_{a' \epsilon A} Q(s', a') - Q(s, a)]$
8: s ⟵ s′
9: **end while**

Algorithm 2: Q-learning(π)

task. One could design a 'good' auto-scaling policy by creating a set of rules [64] and dynamic thresholds [99]. But changes in controlled system such as application updates or modifications in workload patterns require redesign of the policy. There is a need for more sophisticated techniques that can adapt to the changes.

3.4.2 REINFORCEMENT LEARNING

Reinforcement learning [81] is the process of learning where a learner actively interacting with the environment to achieve certain goal. For every taken action the agent receives two types of information: current *state* of the environment and *reward*, which depends on the task and its goal. The objective of the agent is maximize the reward and determine the set of actions (or policy) to that achieve the objective. In case of auto-scaling system the agent learns the target application behavior by taking resource scaling actions and observing response from the application. The application scaling problem can be described as Markov decision process (MDP), which is defined by:

- a set of states S, *can be infinite*
- a start state $s_0 \epsilon S$
- a set of actions A, *can be infinite*
- a transition probability $Pr[s'|s, a]$
- a reward probability $Pr[r'|s, a]$

The model is Markovian, because the transition and reward probability is the function of the current state s and does not depend on the history of states and actions taken before.

Reinforcement learning has a set of learning algorithms, which belong either to the family of *model-free* or *model-based* approaches. The *model-free* approach consists of learning an action policy directly, while *model-based* approach consist of first learning the environment model, and use that to learn a policy. Most of auto-scaling systems that apply RL approach use Q-learning and its modification SARSA algorithms. The algorithms belong to the family of *model-free* approaches, which makes them applicable for online system identification.

The policy in the algorithms is based on Q(a, s)-value function. Each Q(a,s)-value gives an estimation of future cumulative reward when action a in executed in state s. Usually Q(a,s)-values stored in a lookup table. The table maps every state $s \epsilon S$ to it best actions $a \epsilon A$ and corresponding Q-value. The popularity of the algorithms is explained by the fact that they estimate Q-value function in case of unknown model. The Q-learning algorithm pseudo-code presented in 2.

The algorithm starts from initialization phase where each Q-value is set to zero. After the initialization the agent observes first state s. Then from the current state s it selects an action using policy π derived from Q. Usually the agent follows ε-greedy policy to choose the action. It means that with low probability ε the agent takes random action and the rest of the time it takes the action that gives highest expected reward. The random action selection is required to explore other actions that are not taken so far. After the transition to the new state s' (lines 5 and 6) the agent obtains reward. Afterward the Q-value is updated based on received reward and maximum Q-value among all possible actions from the new state s' (line 7). The update rule contains two parameters α and γ. Each of them set between 0 and 1. The first parameter is a learning rate, which defines the learning speed. α can be fixed value or it is a function of a number visited states. The second parameter is the discount factor. It controls the impact of future rewards. Usually the value of 0.8 is used. Finally, the new state s' is assigned as a current state of the agent (line 8). The SARSA algorithm is similar to Q-learning. The only difference is that SARSA uses same action selection policy (line 4) to update Q-value (line 7). In Q-learning the policy is updated based on maximal Q-value ($argmax_{a' \epsilon A} Q(s', a')$).

To apply reinforcement learning approach one has to define a set of actions, state-space and reward function. The choice of state and actions depends on type of scaling horizon. The state-space for horizontal scaling is defined by the number of VMs and input workload. The actions of horizontal scaling are defined as add VM, do nothing, remove VM. Dutreilh et al. [46] extend the set actions by allowing to add and remove number of VMs instead of only one VM.

The state-action space for vertical scaling considers individual resource assigned to a VM. Rao et al. [118] and Xu, Rao, and Bu [150] used CPU time, number of virtual CPUs and memory to define the state. In [116] authors define state as utilization level of each resource. Similar to horizontal scaling the action space for each resource consists of add, remove, do nothing actions.

The reward function determines the optimization goal of the auto-scaling system such as cost of assigned resources, utilization levels and penalty for SLA violation. If SLA is violated then reward can be negative [118]. It encourages the agent to avoid under-provisioning. In addition one could also motivate the agent to take cost-effective decisions by giving higher reward for actions that require to allocate less resources.

As we mentioned earlier, RL offers online model-free learning algorithms that can adapt to application and input workload changes. However, to apply RL we need to address set of challenges. There are three main problems: long learning time, large state-space, exploration-exploitation dilemma.

Long learning time In the beginning of training process decisions provided by auto-scaling system is far from desired ones. The Q-learning algorithm requires significant time [20], before it can provide valuable decisions. Hence, during initialization the actions made by the agent can lead to bad scaling decisions such over-provisioning or even under-provisioning. Therefore to address the issue different approaches have been proposed. Barrett, Howley, and Duggan [20] perform horizontal scaling of web application front-end tier and apply parallel learning that allows multiple agents to exchange learning policy, so Q-learning algorithm converges quicker to optimal policy. Tesauro et al. [132] used data collected from offline-training to initialize Q-table. In [118] authors apply neural-network to reduce time to obtain initial policy. In chapter 5 we present speed up technique for vertical scaling.

Large state-space The grows of state-space dramatically increases a search space for the agent. Especially it becomes challenging if RL is applied for vertical scaling. From one side vertical scaling provides fine-granular resource allocation; from another side the number of states significantly increases with greater granularity. For example, if the state is defined by 2 resources (CPU and RAM) and each resource has 4 possible variations ({1, 2, 3, 4} CPUs; {256, 512, 768, 1024} MB of RAM), then total number of states is $4 * 4 = 16$. If we increase granularity of each resource by factor of 2, then the number of states increases by factor of 4 and

becomes $8 * 8 = 64$. The number of states also increases if we add one more resource with same number of possible values. In case of 3 resources (to control CPU, RAM and I/O allocation [116]) the number of states is $4 * 4 * 4 = 64$. Moreover, the number of states grows if the state-space describes resources assigned to more than one VM. In [118] the state is defined by resources allocated to 2 VMs. To reduce the state-action space authors limited number of possible states and actions for each resource. However, the reduction of states leads to coarse-grain resource allocation. In [116] authors propose to use separate state definition for each VM and apply feedback from reward to resolve resource conflicts. In chapter 6 we present an approach to build MDP model of individual VM resource in isolation from other resources.

Exploration-exploitation dilemma. Q-learning is a model-free algorithm. The agent needs to perform exploration actions in order to observe all states of the environment. However, how the agent can judge whether the obtained policy is optimal in a given environment? Should the agent explore other states or continue to exploit current policy? Usually to resolve the problem the following heuristic is exploited. The agent starts exploration and later on the number of the exploration driven actions diminishes by moving towards exploitation actions. It is also common to fix the exploration rate ε as small value below 0.1 [118, 20, 116]. One of the important parts of exploration is to avoid performance degradation of the controlled application. Usually in the literature the guided exploration is used. For example, in [25] authors during the exploration phase exclude actions that can impact the performance. For example, if the current CPU utilization is high, then actions that scale down resources removed from exploration actions set.

Most of applications are not designed for resource elasticity. Therefore it is necessary to adapt internal application parameters such as *number of threads per session, maximum number of database connections* to gain the advantage of changed resource capacity. The impact of actual parameters values to the application performance is significant [60]. For example, Apache web server has *MaxClients* parameter, which sets the limit on maximum number of requests to be served simultaneously. The small value leads to low resource utilization, while the high value can bring the application to the overloaded condition. In [150, 61] authors exploited RL to tune the parameters. The results presented in the papers show that adapting application parameters improves the application performance. However, the choice of parameters is done manually. An application may have a number of parameters that can drive its performance and resource usage. Dynamic parameter identification might be one of directions for future research.

In conclusion, reinforcement learning is a promising technique for auto-scaling systems. It provides model-free learning algorithms that can adapt system model online. Hence, cloud user does not need to define scaling policy parameters upfront. However, it is important to note that RL has challenges that need to be addressed.

3.4.3 CONTROL THEORY

Control theory provides automation mechanisms for management of complex information systems. Systems under control of feedback loops can deal with disturbances, uncertainties, unpredictable changes. In figure 2 presented a standard feedback loop. The system under control is called *target system*. It has a number of metrics, which marked as *measured output* in the figure. The system has set of control knobs(*control input*). The task of the controller is to periodically adjust control knobs to insure that *measured output* meets its desired value (*reference input*) specified by the system designer. To provide high level of control the controller algorithm should consider *control error*, as well as external *disturbance*, which can impact *measured output* of the system.

Control systems can be *open* or *closed*. First type does not use feedback to verify whether the output achieved desired state. It means that system does not monitor the output of the process it controls. Therefore open loop systems cannot correct control errors and compensate

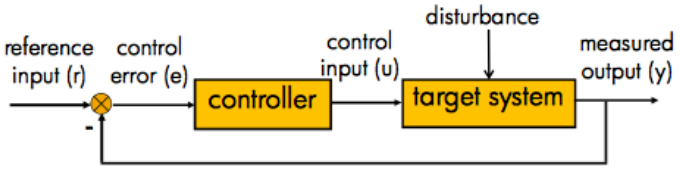


Figure 2: Standard feedback control loop Picture from [160]

disturbances. Usually *open* control systems used for management of simple processes, where the feedback it not an issue. *Closed* systems use feedback. Hence, they observe the output and correct the output if it deviates from the desired value. For the control systems it is important not only react to deviation of the output, but also anticipate the errors. *Closed* systems that predict errors are called *feed-forward*. The best quality of control is achieved when feedback and feed-forward controller work together. We consider only *closed* loop systems, because they offer feedback mechanism that informs the auto-scaling system about the application state and virtual resources utilization. Based on the number of input and output parameters controllers classified into SISO (single input single output) and MIMO (multiple inputs and multiple outputs) controllers. For example, to control CPU and RAM resource allocation process the auto-scaling system requires MIMO controller.

According to Hellerstein, Singhal, and Wang [66] the design of closed loop controller consists of three main steps. Firstly, one should define the control objective. For example, the objective of auto-scaling system is control quality of service by adjusting resource allocation to ensure that performance indicators such as 95% percentile of the response time meets SLO. In this case the *reference input* is specified, so the control system solves regulation problem. There are other examples of the control objectives such, management of resource utilization [108]. The authors target 80% CPU utilization of a web server.

Second step is describing the software system in terms of control theoretic concepts. In figure 2 presented key elements of feedback control system for regulatory control. Assume we want to design auto-scaling controller for a VM running a web server. As the *reference input* we can use the web server response time. The *target system* is VM that runs the web server. The *measured output* is response time of the server. Virtual CPU speed is *control knob* for the response time regulation. The relationship between input and output can be affected by *external disturbance* such as the web server clients request rate. The goal of the controller is adapting the *control input* to keep the *output* consistent with the *reference input* in presence of *external disturbance*.

Third step is obtaining the model to describe the relationship between input and output. In control theory the step is referred as system identification process. The relationship can be derived with help first-principles [65]. Often the exact form of the relationship is not available. In this case, black-box approaches are used to construct the generic models with the help of statistic techniques.

Patikirikorala and Colman [112] provide classification of well-established control schemes. They classify the schemes in four categories: fixed gain controllers, adaptive controllers, model predictive controllers and reconfiguring control.

Fixed gain controllers are the simplest types of controllers. The tuning parameters of the controller are set during system identification experiments. One of the most used controllers is Proportional Integral Derivative(PID) controller. The following algorithm describes PID the controller:

$$u_k = K_p * e_k + K_i \sum_{j=1}^{k} e_j + K_d * (e_k - e_{k-1}) \quad (1)$$

u_k is input value, for example, CPU power of a VM. e_k is a control error, which is calculated as difference between *measured output* y and *input reference* r. K_p, K_i, K_d are proportional, integral and derivative gain parameters. During setup process of the controller the gain parameters of each component(proportional, integral and derivative) are tuned to achieve desired control quality. They don't change during runtime of the system. Therefore the controller is called *fixed gain*. This type of controllers is useful for the systems where workload conditions don't change or change within nominal range. However, if the workload is characterized by high fluctuations, then the controlled system will experience performance degradation. In [96, 45, 95, 68] fixed gain controllers are applied for dynamic resource allocation. Lim et al. [96] build proportional integral controller to allocate application server VMs based on CPU utilization. The workload was changed within predefined operational range using step function. In [95] authors apply integral controller perform horizontal scaling of storage tier (VM running HDFS cluster). The controllers performs reactive scaling when unpredicted load spikes occur. Heo et al. [68] build CPU and memory controllers to scale web server VM. The controller periodically adjusts the resources with respect to changed workload.

Adaptive controllers address some limitations of fixed gain controllers. The controller is equipped with online estimation techniques such as least square method. With help of the technique the controller can tune own parameters to meet user provided high-level objectives. Padala et al. [109] propose adaptive controller for provisioning multi-tier web applications. The controller adjusts CPU and Disk I/O resources of each tier VMs. Authors apply recursive least square method to periodically update the controller parameters. In [108] adaptive controller applied to keep CPU utilization of web application close to 80%. Ferguson et al. [51] use MIMO adaptive controller to meet job deadline of MapReduce application. Authors consider the case where job deadline can be modified. To adapt to changed job deadline the controller dynamically reassigns number of MapReduce tasks running in parallel.

Model predictive controllers. Two previous types of controllers are reactive controllers. They cannot anticipate future behavior of the system. In contrast, the MPC can predict future behavior and perform optimization with respect to predicted value. For auto-scaling systems it is important to have proactive component in order to provide better scaling decisions [17]. Roy, Dubey, and Gokhale [120] apply ARMA based workload prediction and include the workload component to the control loop that adjusts the number of running VMs to maintain user-defined response time. Nathuji, Kansal, and Ghaffarkhah [106] developed MIMO controller to regulate resource allocation between multiple batch applications and provide performance with respect to different quality of service levels.

Reconfiguring controllers Adaptive controllers dynamically adapt the parameters of the controller. However, the control algorithm remains unchanged. Reconfiguring controller is a form adaptive controller that can change control algorithm during runtime. There have been attempt [129] to apply the controller in resource allocation process. However, it lacks stability proofs.

One of the complex parts of applying control theoretic approaches is building the model of relationship between input and output. Classical PID controllers consider single linear models. However, most of inter-relationships in computing systems are non-linear. ARMA(X) (autoregressive moving average) is able to capture the correlation between current output of the system and recent control inputs. ARMA-based models can anticipate future output values and improve quality of control. In [160, 109] use ARMA model to manage resource allocation of web application. Kalyvianaki, Charalambous, and Hand [83] applied Kalman filters to control CPU allocation to 3-tier web applications. They build MIMO model that catches resource usage correlations between the tiers. A number of works[91, 117, 151] employ *Fuzzy models*. The fuzzy model consists of a set of rules which connect input variables with output variables. The model associates workload (input) with resource demand (output). With the help of fuzzy rules input and output of the controller mapped to a fuzzy set. Basically the rules embed human

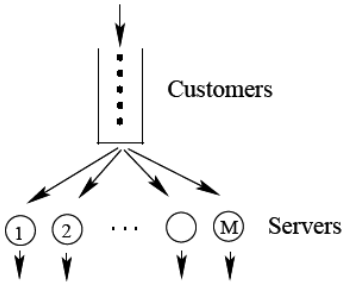


Figure 3: Queuing model from [93]

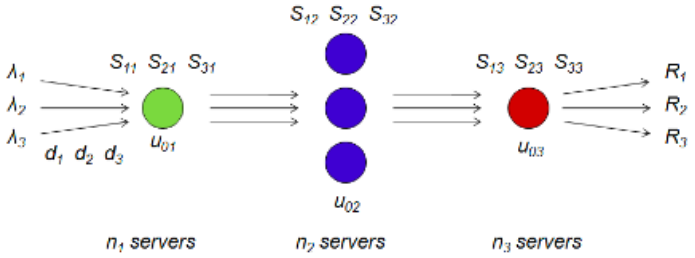


Figure 4: Queuing network from [53]

expert's knowledge. Fuzzy rule describes membership function that determines a value in the range from 0 to 1. Xu et al. [151] developed fuzzy controller to learn relationship between workload, application performance and resource usage. Then obtained model is used to estimate required CPU for the incoming workload. Usually fuzzy model is fixed at design time, therefore the workload with abrupt changes can lead to control overshooting. To address the issue Rao et al. [117] designed self-tuning fuzzy controller that can dynamically correct control overshooting. Authors adjust database VM virtual CPU cap to target desired response time of web application under the workload with high fluctuations.

In summary, control theory provides feedback control mechanism that adapts to workload changes and operating conditions. However, the quality of control greatly depends on the applied model. Many applications have non-linear relationship between performance and resource consumption. Hence, there is a need to apply non-linear models. Unfortunately the control theory does not provide general methodology to obtain the model. The model usually obtained empirically, which requires extensive experiments and deep domain knowledge. Moreover, the accuracy of resource allocation depends on the type of chosen controller. The reactive controller is simple to implement, but it cannot anticipate future needs. Therefore the focus should be on applying model predictive controllers that can provide better scaling decisions.

3.4.4 QUEUING THEORY

Queuing theory based models are common approach to estimate application performance and status metrics such as response time, length of queue and requests waiting time. The models have also been successfully applied for resource allocation and application scaling problems.

The general representation of queuing model is drown in 3. Customers arrive to the system with certain mean arrival rate λ and M processing nodes(application servers) serve them with a mean rate of μ. The common way to characterize queuing model is to apply Kendall's notation [84]. The notation is used to describe and classify queuing model. The model is described as follows: A/S/c/K/N/D. Below presented the meaning of each value.

- A - inter arrival time distribution
- B - service time distribution
- C - number of servers
- K - system capacity. Maximum number of customers allowed in the system. If the limit is reached, then further arrivals are not allowed. If the value is not present, then the capacity is assumed to be unlimited
- N - calling population. The size of the population from which the customers come. If the value is omitted, then the population is infinite.

- D - the Service Discipline or priority order. It defines how jobs are served in the system. Two most used disciplines are FIFO and LIFO.

The last three values (K, N, D) often are not shown. If they are not present, then $K = \infty$, $N = \infty$, $D = FIFO$. Usually only arrival time (A) and service time (B) distributions are considered. Ofter they marked as M, D and G. M stands for exponential distribution. D - means deterministic distribution. If the G is present, then it refers to Gaussian or normal distribution.

The queuing model presented in figure 3 is one of examples of an application that can be scaled horizontally. The queue of customers can be considered as a loadbalancer, which redistributes requests among M servers running inside VMs. To model n-tier application the model has to be extended, so it becomes queuing network as it shown on figure 4.

Most of the works apply two queuing models: M/M/1 and G/G/1. The first model assumes that arrivals λ and service rates μ follow Poisson distribution. In this case the response time can be calculated as follows $R = \frac{1}{\mu - \lambda}$. The second model assumes that normal distribution governs service rate and arrival rate. In this case arrival rate calculated in a following way: $\lambda \geq [s + \frac{q_a^2 + q_b^2}{2(R-s)}]^{-1}$. For multi-tier applications presented on figure 4 the end-to-end response time is a sum of response times provided by each tier [158].

Researchers usually evaluate two types of interactive applications: single tier and multi-tier applications. Ali-Eldin et al. [8] [7] applied G/G/n model to perform horizontal scaling. n - in the model stands for number of servers of the application. They design adaptive proportional controller which reacts to changes in the load dynamics and targets the application performance SLO. The controller has predictive component that anticipates future workload. However, the evaluation presented in the paper is based on simulation, so it is hard to judge whether the controller will be able to provide same quality of control for the real application. In [144] authors analyze actual traces of request arrivals to the application tier. They observe that arrival process follows Poisson distribution. Based on the observation they build provisioning scheme to scale application tier of multi-tier web application. Unfortunately both works target only single tier, while many web applications are multi-tiered.

Zhang, Cherkasova, and Smirni [158] evaluated multi-tier web application. Each tier is considered as queue. Authors empirically found the correlation between request rate and CPU utilization. They proposed a controller that uses regression to approximate CPU usage based on request rate. In [62] authors developed auto-scaling system which performs horizontal scaling of multi-tier web application. The system aims to provide desired response time with minimal number of VMs. Each tier modeled as G/G/n queue. The approach requires offline service time profiling of each tier. During runtime the proposed system classifies workload to decide which tier to scale. For example, 'ordering' requests put more load on data tier, while 'browsing' requests utilize front-end tier.

Many web applications characterized by high dynamics of incoming workloads. Urgaonkar et al. [136][135] combined predictive and reactive methods to provision multi-tier application with respect to incoming workload. To perform capacity planning and respond to flash crowds or deviations from expected long-term behavior it is better to use the combination of two methods. Each server of a tier modeled as G/G/1 queue. The prediction mechanism uses histograms to anticipate peak loads. However, peak load provisioning leads to high resource wastage. Gandhi et al. [53] build auto-scaling system based on queuing-theoretic model that dynamically resizes application tier of a web application to meet user specified performance goal. With help of Kalman filters the system dynamically learns the model parameters and proactively scales the application.

All presented above works exploit horizontal scaling to deal with workload changes. Vertical scaling of front-end tier server was investigated by [38]. One of the issues of e-commerce applications is scalability of database tier that was discussed in section 2.2. Authors propose M/M/1 queuing model to simplify scaling of database tier. They show that single large VM

delivers lower response time and higher throughput in comparison to multiple small VMs, even though there is the same total amount of resources (CPU, RAM) in both systems.

It is common to apply queuing model to scale tiered web applications. However, there are works that address scaling of batch applications. N. Bennani and A. Menasce [105] propose multi-class queuing network to provision batch applications. Authors implemented horizontal scaling of batch application servers based on predicted workload. However, the proposed system does not consider workload dynamics such as change from CPU-intensive jobs to I/O-intensive jobs. The parameters of the model are fixed at design time.

The queuing theory usually applied to the systems with stationary characteristics such as constant arrival and service rates, user think time. However, many cloud applications facing workload fluctuations [82, 119], so the arrival rate can change dramatically. It means that auto-scaling system will not be able properly provision the application when one of the parameters changes. Therefore, the queuing model of the application requires periodical reevaluation, which usually done offline.

3.4.5 TIME SERIES

Time series is common technique to analyze sequence of data points. One of the examples of time series is amount of requests issued by clients of a web server per second. With respect to dynamic resource allocation problem time series analysis is used to find repeating patterns in workload or in resource usage traces. The result of the analysis is employed to predict future workload or resource demand. In order to perform prediction one can use moving average(MA), auto-regression(AR), auto-regressive moving average (ARMA), or machine learning methods such linear regression or neural networks.

Moving average. The predicted value x_{t+1} is based on the average of the sum of previous q values. In general formula for MA looks as follows: $x_{t+1} = x_t * a_1 + x_{t-1} * a_2 + .. + x_{t-q+1} * a_q$. Values $a_1, a_2, .., a_q$ are weighed factors. The sum of factors must be 1. In case of moving average all the factors are equal to the value of $1/q$. For WMA(weighted moving average) the factors values decay from the most recent measurement to the oldest data. Hence, newly arrived data is consider as more important than the old one. Shen et al. [126] used WMA to calculate padding value for virtual CPU cap. In [56] authors show that applying moving average based method to predict CPU utilization leads to performance degradation of the scaled application, because the actual workload is smoothed by the method. Therefore they propose to use pattern detection techniques such FFT(Fast Fourier Transformation). In [111] moving average is applied to smooth job progress measurement noise.

Auto-regression method has been applied in many works [56, 79, 28] that address resource and workload prediction. Formula for AR method looks as follows: $x_{t+1} = x_t * a_1 + x_{t-1} * a_2 + .. + x_{t-p+1} * a_p + \varepsilon_t$, where p is number of previous values and ε_t is a white noise. The quality of prediction greatly depends on the coefficients $a_1, a_2, .., a_p$. Therefore it is important to correctly estimate the values. There are a couple of approaches to calculate the coefficients such as leas squares, methods based on calculation of auto-correlation coefficients, Yule-Walker equations. Jiang et al. [79] track client requests to the web application and predict request rate using AR method. Then predicted value is fed into queuing model to estimate number of required VMs. In [56] authors evaluate AR method resource prediction quality. It outperforms moving average methods. However, the overhead of AR in terms of computation time is high. Similarly Chandra, Gong, and Shenoy [28] apply first order AR to predict application workload.

Auto-regressive moving average (ARMA) it is a combination of AR and MA methods. AR part is similar to previously described method. While MA is a sum of mean and errors terms: $x_t = \mu + \varepsilon_{t-1} * a_1 + \varepsilon_{t-2} * a_2 + .. + \varepsilon_{t-q} * a_q$. The method is well fit to stationary processes. The processes characterized by constant mean and variance of the time series, which do not change over the time. Hence, ARMA should not be applied to the time series that have some trend,

such seasonal variations. For non-stationary processes one can use extensions of ARMA, such as ARIMA and ARMAX. ARMA model was applied by Fang et al. [50] to predict CPU usage of single tier web application. The predicted value is used to vertically scale VM. Authors scale RAM and number of VCPU on the VM. For bursty load authors scale number of VMs. Horizontal scaling is triggered based on fixed upper(80%) and lower(40%) thresholds of CPU utilization.

Some authors employ pattern matching and identification techniques to discover patterns in resource usage [126, 56] or find similarities between Map-Reduce jobs [92]. Shen et al. [126] use signal processing techniques such as FFT (Fast Fourier Transformation) to extract burst patterns in CPU and memory utilization traces and calculate padding value for each resource. However, for every application they evaluated only one resource was scaled. Nguyen et al. [107] applied *wavelet transform* to decompose a signal (CPU usage traces) into a set of wavelets at increasing scale. Then they synthesized the prediction of the original signal by adding up the predictions of these decomposed signals. Proposed approach improves prediction quality in comparison to AR and FFT based prediction techniques.

Neural networks (NN) belong to a family of statistical learning models that were inspired by biological neural networks. It is common to use NN to approximate functions that depend on a number of inputs. Neural network consists of group artificial neurons that connected to each other. The neurons compose a set of layers: input, output and hidden. The number of neurons in the input and the output layers depend on particular problem. For n-sized history window NN has n-neurons in the input layer and one neuron in the output layer, which provides predicted value. In [76] authors apply NN to perform 12 minutes ahead prediction of CPU usage. The size of the interval is chosen with respect to VM launch time (usually 5-15 min). Some authors apply NN in combination with other techniques. Rao et al. [118] use NN together with reinforcement learning. Authors exploit NN to perform approximation of Q-value function. In [88] NN was used to address VM sizing problem. In particular authors discover non-linear relationships between the application resource capacity (CPU, memory, disk I/O) and its performance. Neural networks are efficient modeling and approximation tool. However, to achieve high accuracy one needs to determine the structure of NN (number of neurons in each layer, activation function). Moreover, the size of training data also plays integral role in determining the accuracy of NN.

Time series analysis provides wide range of techniques to predict future resource usage or incoming workload. It is common to apply time series for predictive resource allocation. However, the quality of prediction depends on the application workload, chosen prediction technique and its parameters such as history window, prediction interval. Time series techniques can be combined with other reactive techniques to perform resource and workload anticipation.

3.5 SUMMARY

Auto-scaling process consists of four phases: monitoring, analyzing, planning and execution. In this chapter we focus on analysis and planing phases. The auto-scaling system uses the analysis to answer the question *when* to scale and the planning phase to make decision about *how many* resources to allocate. There are different techniques that can be used to implement the analysis and the planning phases. Generally, they can be classified into five categories: threshold based techniques, reinforcement learning, control theory, queuing theory and time series analysis. Threshold based technique allows to perform only reactive scaling. Most of public providers offer auto-scaling services based on this techniques. The approach became popular due to its simplicity. However, to design a good scaling policy one needs to determine optimal scaling rule parameters. In contrast to the previous technique time-series analysis offers wide range of methods for prediction. The accuracy of predicted value depends on chosen method and the parameters values of the method. Time-series analysis provides the tools for proactive scaling that anticipate future resource demand. Hence, we can increase the effectiveness of resource allocation. Queuing theory and control theory based techniques present two auto-

scaling methods that rely on modeling the system. Queuing theory usually models relationship between requests arriving and leaving the system. A main limitation of queuing theory based auto-scaling systems is that the model is fixed on design time. Therefore any changes either in the application or in the workload require updating the model. Similarly, to queuing theory, control theory depends on the application performance model. The theory offers a wide range of controller design approaches that can be used to implement reactive and proactive auto-scaling systems. Moreover, some controllers have self-adaptation capabilities. However, the performance of the controller highly depends on the application model defined during design time. The last technique is based on reinforcement learning approach. In contrast to queuing and control theory it does not require *a priory* knowledge or model of the application. Instead, it learns to make optimal scaling actions by taking trial-and-error approach. RL seems appealing technique for auto-scaling systems. However, one needs to address state-space complexity and long-learning time challenges.

VERTICAL SCALING FOR PRIORITIZED VMS PROVISIONING*

*The contents of this chapter first appeared at CGC'12 [153].

4.1 INTRODUCTION

Cloud became a popular computing platform, which offers on demand computing resources and storage capacity. IaaS providers deliver fixed size virtual resources in the form of VMs. Each VM has fixed amount RAM and CPU, which do not change over the VM lifetime. Cloud users can select a VM from the set of offered VM types. For example, Amazon has *small*, *medium* and *large* VM types. Resource demand of many applications is not static and varies over the time. To achieve high application performance users are forced to acquire VMs based on the application peak demand. However, peak load resource allocation leads to resource wastage. As a result users pay for resources that have not been utilized.

To improve utilization users can acquire smaller VMs with the help of cloud auto-scaling controller. It allocates a VM when the application resource demand is high and de-allocates the VM when the demand is low. Cloud providers such as Amazon [9], RightScale [125], Scalr [123] offer threshold based controllers. However, the user needs to define controller parameters, which significantly impact scaling policy. One of them is scaling threshold, which is used by the controller to decide when to trigger scaling action. For most of the users the process of defining the threshold is not an easy task. It requires understanding of the application resource usage and experience with threshold based controllers. Moreover, due coarse-grained resource provisioning model (fixed size VM assignment) it is hard to closely follow the application resource demand.

Recent works [41, 143, 124, 119] analyzed cloud and data-center workloads and found that applications running in the cloud can be classified into two groups: latency sensitive interactive applications and batch applications. The first group consist of web applications such as internet stores, bookkeeping sites and etc. Low latency performance is an essential requirement for these applications. None of the application users would like to interact with a slowly responding web site. For e-commerce applications high latency means potential revenue loss, because users will eventually move to more responsive web sites. The second group consists of applications running as back-end tasks such as MapReduce jobs. Batch applications do not require real time responsiveness and can tolerate performance slowdown.

Exiting cloud provisioning model supports only horizontal scaling (adding and removing VMs). Alternatively, we propose to enable vertical scaling. In contrast to horizontal scaling, vertical scaling allows to modify VM size on-the-fly. For example, adding more CPU power to a running VM or changing RAM size. Vertical scaling is beneficial for both sides of the cloud market. From one side user running web application can really follow the application resource demand curve and pay for resources actually used. From another side cloud provider can run on the same host batch application of other user, which would utilize remaining resources on the host. Such collocation can potentially improve utilization of the cloud and lower energy costs.

To run different classes of applications on one host cloud provider needs to prioritize VMs. Prioritization allows interactive application to rent capacity from batch application during resource contention. Moreover, batch application is able to harvest residual resources from high priority VM. Amazon *spot instances* is one of the examples of resource harvesting. Users bid for resources left from high priority *on-demand* and *reserved* EC2 instances. The lifetime of the *spot instance* is not guaranteed. It runs until Amazon has enough unused capacity. The instance can be terminated any point of time. It means that application running inside *spot instance* should be fault tolerant. However, fault tolerance does come for free. It always comes with recovery overhead. One needs to recover application state from last checkpoint by reading data from persistent storage.

In this work we develop resource allocation controller that performs vertical scaling of collocated VMs. We build the controller on top of popular Xen hyper-visor. The controller uses VM CPU usage traces to predict future resource demand and trigger scaling actions. We evaluate our controller against real world workload traces. The evaluation results show that the controller

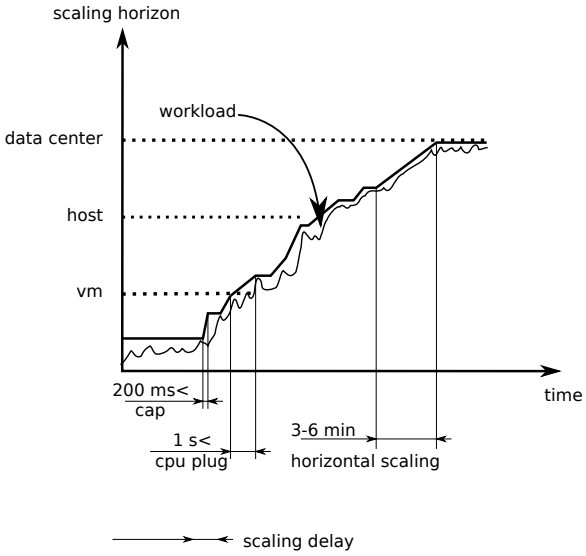


Figure 1: Overhead of different scaling types

provides low latency for interactive application running under highly fluctuated workload. The controller also resolves resource contention conflicts among collocated VMs. It rents resources from low priority VM (without termination), if resource demand of high priority VM exceeds its capacity. As a result, low priority VM makes progress even during resource contention. If high priority VM has residual resources then controller assigns them to the low priority VM.

4.2 DESIGN RATIONALE

Virtual resources scaling can be performed horizontally or vertically. Horizontal scaling means adding a new server replicas or load balancers to distribute load. Vertical scaling means changing on the fly assigned resources to an already running instance. For example, we can modify virtual CPU power of running VM. Most of cloud platforms exploit only horizontal scaling. For cloud provider it is easier to schedule fixed size VMs rather than flexible size VMs. However, the drawback of horizontal scaling is non-ignorable VM instance acquisition time and coarse-grained resource allocation.

There are different numbers about the instance start-up lag [70, 21], but on average it takes about 1 minute. It is unacceptable for interactive applications which have latency requirements in seconds range. In contrast, vertical scaling allows to double VM power in less than a second [47, 110], which makes it attractive for interactive applications.

Modern data centers host large number of applications. Resource demand from the applications varies over the time. Cloud provider use fixed size virtual resources (VMs) to place users' applications. User can acquire and release VMs based on application resource demand. However, if the utilization goes below VM size, then the user has no option to release unused resources. From side cloud provider simplifies VM scheduling problem, from another side such approach leads to low datacenter utilization levels [119] and goes against economical motivation of the user. Users always have to pay for unused resources if the VM is not fully loaded.

Efficient allocation of resources can be achieved with combination of vertical and horizontal scaling. Figure 1 describes our vision of scaling types in virtualized data center. We define following scaling levels: VM, host and data center. Each level has certain range of scaling overhead.

VM level scaling As we mentioned earlier, applications running inside VM might not fully utilize allocated resources all the time. One of examples is web applications, which utilization depends on a frequency of accesses by the application clients. For example, to follow utilization

curve we can tune virtual CPU (VCPU) power. Most of existing hyper-visors support VCPU capping. Hyper-visor can set limit on the maximum amount physical CPU cycles, which a VM can consume. In *Xen credit scheduler* [37] the cap is expressed in percentage of one physical CPU: 100 is 1 physical CPU, 50 is half a CPU, 400 is 4 CPUs, etc. Gong, Gu, and Wilkes [56] claim that changing the limit of a virtual machine CPU takes about 120 ± 0.55 ms. Hence, capping reallocates CPU resources with small overhead. However, VM cannot get more than its maximum allocated resources even if a host has spare cycles on other cores. If the VM has 2 cores, then the highest cap value is 200. To get more CPU power, we need to add virtual CPU to the VM.

Host level scaling If the VM level scaling reaches its maximum, then CPU-hotplug allows to extend the VM capacity further until it hits the host limit. Modern operating systems support CPU hot plug, i.e. a CPU could be plugged or unplugged to and from an already running VM. Recently, Joyent [80] cloud provider announced support of on-the-fly CPU plugging. The feature improves performance of CPU intensive applications. According to the presented benchmarks, CPU burstable VM (CPU is plugged-in to a running VM) essentially outperforms VM provided by EC2. Plugging CPU means that a VM can get remaining CPU of the host. However, if the remaining CPUs dedicated to another VM, then we need to unplug CPU from collocated VM. To enable such action we need to prioritize VMs inside a host.

Data center level scaling Vertical scaling is bounded by the capacity of the host. Hence, we need to trigger horizontal scaling, if resource conflict is impossible to solve inside the host. Horizontal scaling has significant overhead. In best case it is about 1 minute. To decide when to trigger horizontal scaling action one can use long term resource usage prediction. Data center level scaling we leave for future research.

In this work we mainly focus on vertical scaling. We aim to provision interactive application running in collocation with batch application VM. To efficiently perform vertical scaling of the applications we need to answer two questions. The first question is when to perform VM scaling. The second question is how to resolve resource conflicts on the host.

One of the well-known auto-scaling mechanisms is threshold based scaling. The threshold defines upper and lower limits on target metrics. The scaling action triggers when the limit is crossed. Cloud providers such as Amazon [9], RightScale [125], Scalr [123] offer threshold based controllers. In many controllers users have to define the upper and the lower thresholds as well as low level metrics such as CPU or RAM usage. The quality of resource allocation is highly depends on the chosen threshold values. For most of the users the process of defining the threshold is not an easy task. Hence, users need auto-scaling controller which would offload burden of defining resource allocation policy parameters. To determine the parameters the user has to tune the controller offline. Instead of offline tuning, we propose to track VM CPU usage traces and perform one step ahead prediction. In the next section we present the controller architecture.

The conflicts between collocated VMs usually resolved based on VM priority. There are two common solutions. The first is migrating low priority VM to the less loaded host [126]. However, such approach leads to network transfer and possible service slowdown or even non-zero down time. The second approach is shutdown low priority VM. Since 2009 Amazon launched *spot instances*. Users can bid for *spot instances*. However, the instance can be terminated at any point of time. It means, that internally Amazon creates resource room for high priority VMs by switching off *spot instances*. In this work we also consider, prioritized VMs. However, we propose to place two different classes of application on the host and resolve conflicts by scaling collocated VMs vertically. Such approach allows to keep running both VMs.

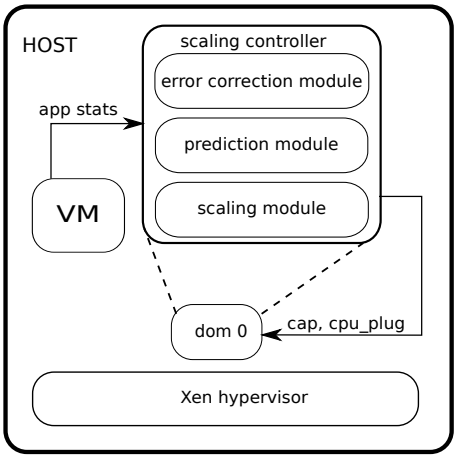


Figure 2: Elasticity controller

4.3 CONTROLLER ARCHITECTURE

To perform resource allocation to the application running inside VM we design scaling controller, which exploits vertical scaling. The controller monitors resource usage of the application and predicts future resource demand to make decisions about VM resource scaling. The architecture of the controller presented in figure 2. The controller runs inside *dom0* of *Xen* hyper-visor and consists of three modules: prediction, error correction and scaling. We use *xentop* resource usage monitor to obtain VM CPU utilization data. Monitoring is implemented with 1 second sampling period. It allows the controller quickly detect changes in the VM resource demand and perform reactive scaling. To perform virtual CPU scaling we use *Xen credit-scheduler* API which adjusts VCPU cap value. CPU plug is implemented using Xens *vcpu-set* API. Unfortunately, CPU hot unplug is not implemented in the current version (4.1) of Xen. In particular it does not remove CPU from OS running inside VM. Therefore we wrote small daemon which implements CPU ejection inside the VM.

The work-flow of the controller runs as follows. Every sampling interval CPU usage values sent to the prediction module. The module uses CPU usage history data to perform forecasting of CPU demand value for the next $t+1$ step. The error correction module adds an extra padding to the predicted value. It is done to avoid under-provisioning when the predicted value is below actual CPU demand. Finally, scaling module sets virtual CPU limit to the VM. If the predicted resource usage value exceeds current VM CPU power, then CPU plug action is triggered. Virtual core is unplugged when predicted value indicates that one of the CPUs will not be utilized.

Prediction module To predict CPU usage we apply one of time series analysis techniques - auto-regression(AR) model. The model specifies that output variable linearly depends on its previous values and stochastic term. Dinda and O'Hallaron [42] analyzed CPU usage prediction quality of AR models. Authors found that AR model order of 16 is the best linear prediction model for CPU usage traces. It can predict CPU usage from 1 to 30 seconds in the future. Another important aspect for us is computational overhead. The evaluated AR model has low CPU cost - less than 10 ms to analyze 2000 samples. In our controller we implement one step ahead CPU usage prediction. Therefore we need less data to feed in to AR model. We use only 100 samples history window to perform prediction.

Error correction module The controller uses *Xen credit-scheduler* cap value to limit virtual CPU power of VM. It means that CPU demand above the value is seen as 100% CPU utilization of the VM. Such situation is called resource under-provisioning. During resource under-provisioning the predictor gets only virtual CPU limit value. Actual CPU demand is not known. Hence, wrong values are used for prediction. To deal with the situation we added to our AR model *prediction error correction* and *under-provisioning error correction* mechanisms. The first

mechanism performs active error correction and the second one proactive error correction.

Prediction error correction The goal of error correction is actively correct prediction errors by adding small extra value to the predicted CPU demand. Let $e_1, .., e_k$ denote as a set of prediction errors. We compute e_i as $x_i^{pred} - x_i^{mes}$, where x_i^{pred} is predicted value and x_i^{mes} is actual measured CPU usage value. We only consider under-estimations. Therefore if $e_k > 0$, then we set $e_k = 0$. We use the set of underestimation errors $e_1, .., e_k$ to calculate *WMA* (weighted moving average). WMA gives higher weight to the most recent values. The error correction module takes $|WMA(e_1, .., e_k)|$ and adds it to the predicted resource usage value.

Under-provisioning error correction The prediction error correction is possible only if x_i^{mes} below assigned virtual CPU cap value. It means that real resource demand is unknown during under-provisioning. We have only lower bound, which is assigned cap value. Hence, the predictor cannot properly estimate future resource demand. To deal with under-provisioning we need to immediately raise CPU cap value. We increase the cap value by $\alpha > 1$ until the under-estimation exists. Fir example, if correct cap value is x and under-estimation occurs. Then cap value for next step is $X * a$. It is true that such scheme can cause over-provisioning. However, the controller prediction model will eventually catch up real resource demand and cap value will be corrected.

The value of $\alpha > 1$ it is a tradeoff between error correction speed and resource wastage. To avoid high resource wastage cloud provider can limit range of possible $\alpha > 1$ values. The *under-provisioning error correction* triggers when relative resource utilization r ($0 < r < 1$) crosses predefined threshold r_{thr} The relative resource utilization r computed as follows:

$$r = \frac{v}{u} \tag{1}$$

Where v is VM CPU usage and u is CPU allocation value, which is dynamically changed by the controller. We use equation 2 to calculate $\alpha > 1$ coefficient.

$$\alpha = 1 + \frac{r - r_{thr}}{1 - r_{thr}} \tag{2}$$

Choosing appropriate threshold allows to provide high performance and avoid wasting of resources. The threshold value is very much dependent on the type of workload. For example, highly interactive and bursty workloads require greater over-provisioning value in comparison to less fluctuate workloads. Hence, for bursty workloads the threshold should be lower than for workloads with small fluctuations. Currently our scaling controller uses static predefined threshold $r_{thr} = 90\%$. Automated detection of the threshold is beyond the scope of this paper and will be part of future research.

Scaling module communicates with Xen provided APIs to perform VM resource assignment. The module takes calculated CPU cap value from the error correction module. The cap value fixes the maximum amount of physical CPU a VM is able to consume. Moreover, the module uses calculated cap value CPU_{cap} to decide whether CPU plugging action is necessary. It applies equation 3 to make appropriate decision about CPU plugging. $VCPUS_{curr}$ is a number of virtual CPUs currently assigned to the VM. Maximum cap value for one virtual CPU is 100.

$$action = \begin{cases} plug, & \text{if } CPU_{cap} > VCPUS_{curr} * 100 \\ unplug, & \text{if } CPU_{cap} > (VCPUS_{curr} - 1) * 100 \end{cases} \tag{3}$$

4.4 EVALUATION

To evaluate our controller we build test bed which consists of one host machine running VMs and web workload generator that composed of client emulator machines. The host machine has quad-core Xeon 2.66GHz, 16 GB memory, 100 Mbps network connection with client machines.

All machines run Ubuntu 11.10. Xen version 4.1 is installed on the host machine. The hosting machine has 3 guest VMs running CentOS 6.0 64 bit. One VM has 3 cores and runs DB server. Another 2 VMs running web server (Apache HTTP 2.2) are batch application are managed by our online scaling controller. Dom0 domain of Xen is pinned to one core and does not share it with other VMs. The web server VM and batch VM share 2 CPUs.

We use RUBiS online auction benchmark [121] to build our web application. It consists of web and database servers. The business logic of the auction is hosted on the web server. Therefore CPU load of the web server is much higher than database server and has large CPU usage fluctuations. The load of database is fairly low. Therefore in our evaluation we scale only web server VM since it has significant CPU load variations. We over-provisioned database VM to make sure that it does not cause resource bottlenecks.

The workload generator of the benchmark consists of 10 machines with 100Mbps network connection that run RUBiS benchmark client emulators. We used web traces of WorldCup 98 [134] to evaluate scaling controller with real workload variation. We apply these traces to RUBiS benchmark client emulator. To evaluate the controller we constructed 15 minutes long trace.

Our evaluation primarily consists of two parts. In the first part we evaluate the controller with configurations and analyze the quality of single VM scaling. In second part of the evaluation we collocate the web application VM with the batch application VM. The controller resolves CPU contention between the VMs.

4.4.1 SINGLE VM SCALING

In this part of the evaluation the controller scales web server VM VCPU. The goal is to compare the controller against static resource allocation scheme and find best parameters for the controller. In the experiments we use following configurations: 1) *Mean alloc*: static CPU allocation which is calculated as mean CPU demand over the full workload trace. We assign 1 VCPU to the VM because the mean value below 100%; 2) *Peak alloc*: static peak load CPU allocation which is maximum CPU demand from the trace. Therefore we assigned 2 VCPUs to the VM; 3) *OnlinePad* we run the controller in fixed padding mode, where it adds constant extra value to the prediction to correct under-estimation error. The extra value is percentage value of the prediction; 4) *OnlineCorr* Controller implements CPU allocation with under-provisioning error correction.

In figure 3 is presented CPU demand trace of RUBiS web server. CPU utilization fluctuates over the time and has bursty load spikes. For example, there is a sudden load spike at 600 seconds. CPU utilization jumps from 100% to 125%. At that point of time request rate jumps from 200 to 400 requests per second. The graph also shows, that workload requires more than 1 virtual CPU, because CPU usage some periods of time higher than 100%. It means that in the cloud environment the user has to acquire a VM with 2 virtual cores. However, from the graph it is clear that 2 cores are never fully utilized. Hence, under exiting cloud pay-as-you-go model user would need to allocate VM with 2 CPUs to meet workload demand. And pay for resources that not used most of the time.

We also analyzed the prediction quality of the AR model on the CPU usage trace. Figure 4 presents cumulative distribution of CPU utilization error prediction. The results show that applied model has less than 10% of predicted values with under-estimation error above 3%.

In figure 5 we draw 95% response time of the server. For better representation we exclude from the graph *Mean alloc* configuration, because it has 2000 ms response time. It is significantly larger than response time provided by other configurations. Peak CPU allocation has the lowest response time 14 ms. Hence, a user that assigned VM according mean CPU demand would get two order of magnitude worse performance. Response time of web server under dynamic allocation configurations at most has only 3 times difference in comparison to the peak

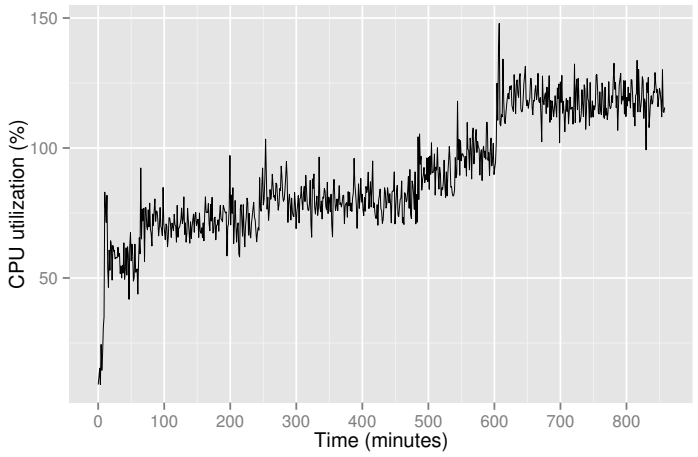


Figure 3: CPU demand of the RUBiS web server

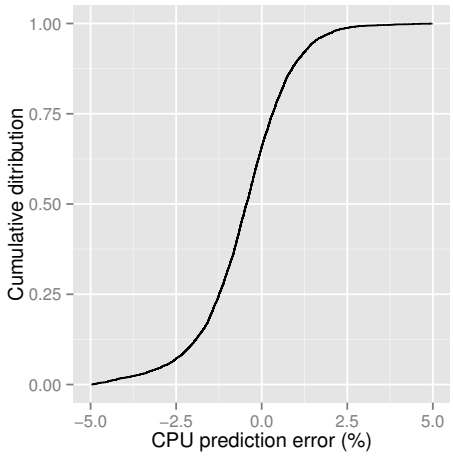


Figure 4: AR model prediction error(e_k)

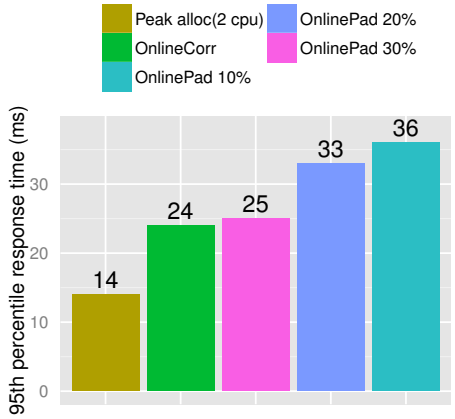


Figure 5: Web server response time

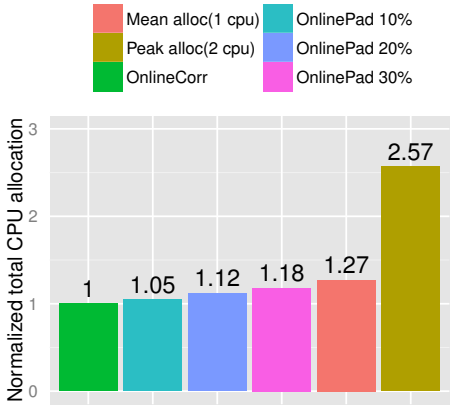


Figure 6: Total resource usage

allocation. Response time achieved by fixed padding schemes increases as the padding value decreases. To obtain the lowest response time one needs manually tune the padding values. The error correction configuration does not require adjustment of the padding value. It uses misprediction value to dynamically compute desired padding value. Among dynamic resource allocation schemes *OnlineCorr* has lowest 95% response time.

Another important aspect of resource scaling is minimizing the cost of acquired resources. In figure 6 shown normalized total CPU allocation for each configuration. We calculated total CPU allocation as a sum of CPU cap values over the run time. We assume pure pay-as-you-go model with 1 second billing cycle instead of commonly used 1 hour cycle. The total CPU time give an estimation of the costs paid by the user in pure as-you-go-model. The controller in *OnlineCorr* mode saves 27% and 157% of CPU time in comparison to static VM configurations with 1 CPU and 2 CPU respectively. The difference between *OnlineCorr* and fixed padding modes varies in range from 5% to 18%. Among the evaluated configurations the controller in *OnlineCorr* mode provides better results in terms of application performance and resource allocation costs. The results of *Mean alloc* configuration show the importance of dynamic resource scaling. Cost and performance of *Mean alloc* configuration is higher than cost and performance of the controller under different configurations.

4.4.2 PRIORITIZED VMS SCALING

In the second part we perform VM collocation on the host. We assume that cloud provider places interactive and batch VMs on the host. The goal of placement is to avoid expensive VM migration if resource contention is detected. Instead, the provider takes resources from low priority batch VM, when interactive application VM exceeds it capacity. In low priority VM we run Apache Hadoop jobs. Hadoop is an open source implementation of MapReduce paradigm [40]. We deploy Hadoop in single node mode and execute *Wordcount* application. In the experiments the VMs share 2 physical CPUs.

Xen scheduler can run in two modes: work-conserving (wc-mode) and non work-conserving (nwc-mode). In wc-mode each VM is assigned a *weight*. In this mode share (weight) is guaranteed. Hence, CPU is idle, only if there is no active VM. For example, if two VMs run on the host and one of them gets blocked, then second one can consume entire CPU. Hence, in Xen's wc-mode batch application should get residual CPU cycles of interactive application. In nwc-mode shares are capped. It means that, in case of two VMs with equal shares, each of them gets 50% CPU, even if second half of CPU is idle. Our controller uses nwc-mode and it decides when to rent or give back resources of low priority VM. Generally we test four configurations: 1) *Standalone VM* : Web server VM runs alone, we do not run Hadoop job; 2) *Xen scheduler default* : VMs have equal weights: ; 3) *Xen wc-mode* : CPU allocation is implemented by *Xen credit scheduler* running in wc-mode, where we give highest weight to high priority VM and lowest to the batch VM; 4) *Controller* : we run our controller in resource error correction mode, because it showed best results among the evaluated controller modes.

Figure 7 shows response time variation of the web application server. Running collocated VMs with equal weights *Xen scheduler default* leads to high fluctuations of response time. Interactive application VM needs to wait until, batch VM finishes its time slice on CPUs. As a result responses are delayed. Changing weight of interactive application to higher value reduces fluctuations. However, it is still higher than response time provided by our controller. By default, credit scheduler uses 30 ms time slice for CPU assignment. Hence, VCPU of each VM gets 30 ms before being preempted. It means that in worse case high priority VM VCPU has to wait for 30 ms before being scheduled. In case of nwc-mode, low priority VM is not scheduled if it runs out of credits. One can notice that during first 100 seconds the response time jitter of interactive application managed by our controller is higher. During the first 100 seconds the controller collects CPU traces for prediction and does not perform scaling. VMs have equal weights. To provide better performance during first 100 seconds, we can use *Xen wc-mode*. After 100 seconds the controller has response time between configurations running two VMs.

Figure 8 shows web server 95% response time of all evaluated configurations. To make fair comparison we took values from 100 seconds to 500 seconds when applications in all configurations competing for CPU resources. The controller runs batch application for 100 seconds (see figure 9). Our controller provides response time which is closest to the single VM mode. The response time provided by Xen credit scheduler in weighted mode is higher by almost 20 ms.

On figure 9 presented the execution time of batch job. CPU allocation implemented by *Xen credit scheduler* in non-weighted and weighted mode provides close execution time value. If we apply our controller then *Wordcount* application runs 2 times longer, than when we use *Xen scheduler*. However, if run time is not critical for low priority VM, then it is a small price paid to achieve stable response time of high priority VM. The graph shows that, wc-mode is better for computationally-intensive workloads, rather than for interactive applications. To provide higher performance for latter ones in wc-mode we need to lower the length of credit scheduler's time slice. However, it can increase the overhead of context switching and reduce effectiveness of CPU cache. Alternatively, the higher performance can be achieved in nwc-mode.

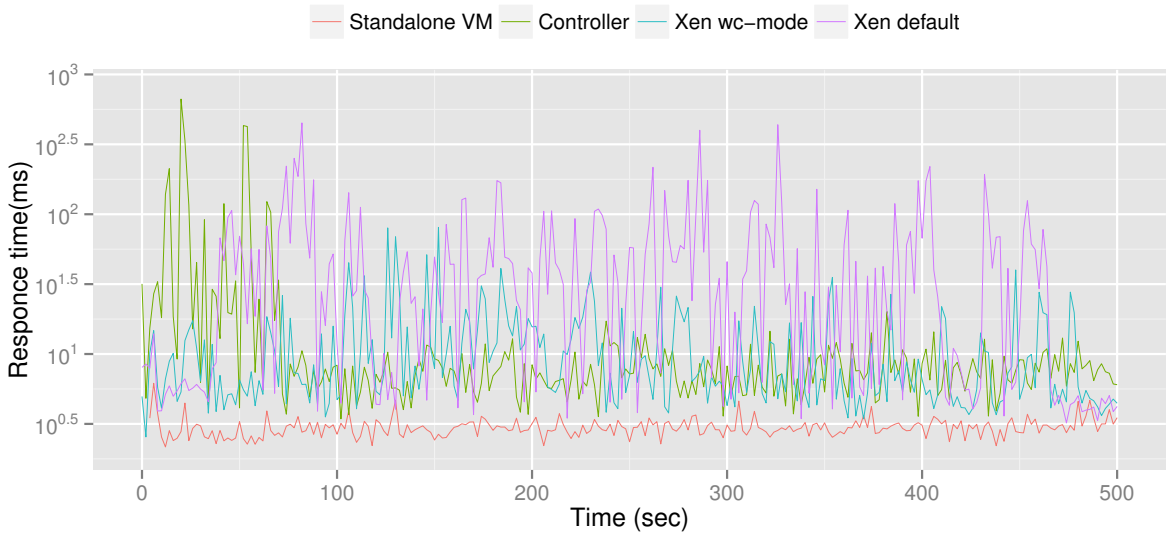


Figure 7: Web server response time

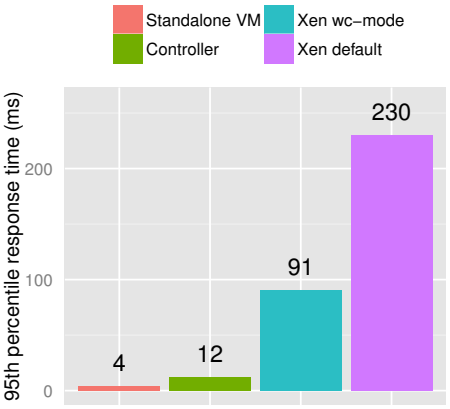


Figure 8: Web server response time from 100 to 500 seconds

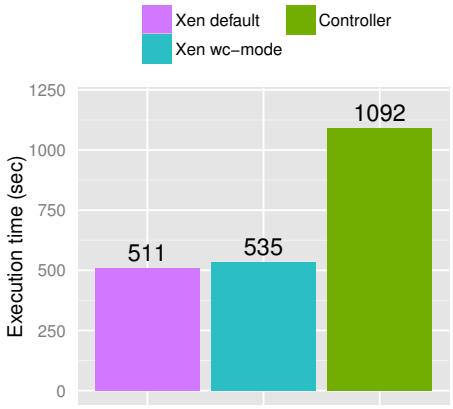


Figure 9: Hadoop execution time

4.5 DISCUSSION

Amazon EC2 Spot Instances provides access to virtual resources for a lower price. The provider some periods of time has spare resources, which are not subscribed by normal *on-demand* and *reserved* instances. To utilize these resources it offers them in the form of *spot instance*. *Spot Instances* have lower price in comparison to *on-demand* instances, because execution time of *spot instance* is not guaranteed. The provider needs to shut them down to leave room for on-demand and reserved instances. However, such approach forces *spot instance* users to run only fault tolerant applications, otherwise the application state will be lost.

We think that vertical scaling can relax fault tolerance requirements to applications running inside low priority VMs. Cloud provider can scale down low priority VM and scale up high priority VM. There is no need to shutdown low priority VM. We assume that users choose VM priority upfront during deployment process. However, the question is how to account allocated resources. The resources of a VM in this model can change at any point of time. Therefore to account resources allocated to the VM during run time, provider needs to shift billing cycle from hour granularity to a second granularity. Then the cost of resources assigned to a flexible

size VM can be calculated as follows:

$$P_{total} = C * \sum_{i=}^{T} r_i \tag{4}$$

Where C is the cost of one hour of full CPU time of the VM and r_i is CPU capacity assigned to the VM during i-th second of the VM execution time.

Applying presented cost model creates following service. Suppose a high priority VM which costs 0.3$ per hour and a low priority VM with price of 0.05$ per hour. VMs run for 2 hours and share 2 CPU of the host. The high priority VM during the period uses 66% of VMs aggregated CPU resources. Hence, the low priority VM gets the rest 34%. The total cost for the high priority VM is $2hours*0.66*0.3\$ = 0.396\$$ and for the low priority VM is $2hours*0.34*0.05\$ = 0.034\$$. Hence, in total cloud provider gets 0.396 + 0.034 = 0.43$

Let's take a look on the current pricing model. Standard *on-demand* VM with one CPU costs 0.1$ per hour. Customer, who wants to run web application and avoid under provisioning, selects VM with 2 CPUs. Hence, total price of running the VM for 2 hours is $0.1\$ * 2CPUs * 2hours = 0.4\$$. In comparison to our model user overpays 0.4 – 0.396 = 0.004$, even if the cost of *on-demand* instance 3 times lower than the cost of flexible size VM. Assume that cloud provider knows that during first hour *on-demand* VM utilizes only 1 CPU. Hence, the provider allows to launch *spot instance*. Suppose that it costs 0.05$ per hour and it was terminated after one hour. Then the user of the instance is going to pay $0.05\$ * 1CPUs * 1hour = 0.05\$$. The cost is 0.05 – 0.04 = 0.01 higher, than the cost of flexible low priority VM in our model. Hence, user of fixed size VM model pays more.

We present here sample numbers. But in general the example shows the advantage of applying vertical scaling to prioritized VMs provisioning. Applying the proposed model can be beneficial for both participants of cloud market. Provider gets an opportunity to acquire more users and better utilize datacenter capacity. From another side users pay for resources that actually being used by the VM.

4.6 RELATED WORK

Most of cloud providers offer semi-automatic scaling systems that scale applications based on resource demand. However, one needs to find optimal parameters for scaling policy. Many research works addressed the problem of dynamic resource allocation. The approaches taken in the works can be classified based on the technique applied in resource allocation system. The most common techniques are feedback controllers [83, 108], regression analysis [44, 126] and queuing theory based analytical models [137, 158]. Our work is complimentary to the previous research. We present dynamic resource scaling controller which driven by AR based prediction model. In comparison to the related work, the controller can handle prediction error to provide high application performance and resolve resource conflicts among collocated VMs.

The problem of conflict resolving addressed by Shen et al. [126] and Lin and Dinda [97] . In [126] authors present dynamic resource provisioning system, which performs vertical scaling of collocated VMs. The system performs resource demand prediction of collocated interactive VMs. As soon as predicted aggregated demand of the VMs exceeds host capacity the system triggers resource conflict resolving mechanism. To resolve the conflict authors set priority to each VM and migrate low priority VM to leave room for high priority VM. Such approach keeps high priority application performance on desired level. But, migration is not an instant process. It means that low priority VM may experience non-zero down time and performance slowdown. In our work we propose to collocate VMs based on application type. Batch application is less sensitive to performance slowdown. Therefore we can avoid expensive VM migration and instead rent resources from batch application to interactive application.

The work closest to ours is done by Lin and Dinda [97]. Authors developed Linux based scheduler to run mix of batch and interactive applications on the host. The scheduler executes each VM VCPU for certain time slice within scheduling period. To prioritize VMs authors give smaller time slice to the batch application and bigger time slice to the interactive application. It means that, the batch VM cannot get spare CPU cycles from interactive VM. Basically authors statically fixed VCPU cap value for each VM to insure performance isolation. In contrast, our controller can dynamically change VCPU cap value to lease utilized CPU resources to low priority VM.

4.7 CONCLUSION

Many application have varying resource demand, therefore the model of fixed size resource allocation, which is supported my most of cloud providers leads to under-utilization. As a result cloud users pay for resources that have not been used. From another side when the number of subscribed VMs reaches datacenter limit cloud provider cannot acquire more users, even if actual datacenter utilization is lower. To improve utilization of the cloud we propose to enable vertical resource scaling and collocate interactive and batch applications which have orthogonal temporal characteristics.

In this chapter we presented resource scaling controller for web applications. The controller dynamically scales VM VCPU power based on the application demand. We apply AR-model to prediction the application CPU usage. We believe that vertical resource scaling reflects interests of cloud users. Since, the application gets resource which it actually uses. The evaluation shows that controller outperforms mean based resource allocation. Moreover provides 2.5 times lower cost and only 10 ms longer 95% response time in comparison to peak based allocation.

The controller also supports collocated VM scaling. At first it tries to satisfy resource demand of interactive application and remaining resources it assigns to low priority VM. The evaluation results show that in comparison to Xen based prioritization mechanism the controller provides higher performance for interactive application.

AUTONOMIC VIRTUAL MACHINE SCALING*

*The contents of this chapter first appeared at CLOUD'13 [154].

5.1 INTRODUCTION

Recent observations by Agmon Ben-Yehuda et al. [4] of IaaS trends state, that the model of fixed bundles, so called "instance types" will eventually change to flexible bundles. The change is mostly economically driven. The reason is that cloud users do not want to rent 6 CPU cores if it is required only 5 of them. Moreover, model of fixed bundles forces cloud users to provision applications with time varying resource demand for peak load. This strategy leads to high resource under-utilization, because average resource demand far below assigned capacity. Hence, users have to pay for resources which are not actually used. Second observation is a size of cloud billing cycles. Most of cloud providers have 1 hour billing period. It means that user has to pay for the whole hour, even if VM was running only for 10 minutes. Existing model does not reflect users' economical expectation of pay-as-you-go model. Authors conclude that IaaS providers will eventually shrink billing periods and allow users to build VM they want to run. The presented trends already exist in the cloud market. Cloud providers such as CloudSigma [33], ProfitBricks [113] and GridSpot [59] deliver virtual resources in the form of flexible bundles.

Many cloud application have varying resource demand. The model of flexible bundles facilitates more efficient resource provisioning for such applications. Users can dynamically resize VMs based on current resource demand. However, to perform efficient resource provisioning(reduce under-utilization and meet application performance goals) one has to design good scaling policy. There is a need for dynamic VM configuration technique.

Most of cloud providers offer auto-scaling services at the IaaS level. The services exploit threshold based scaling approach. The approach tends to focus on scaling at the machine or VM level. But it does not facilitate the definition of higher business function, such as user specified QoS. Using threshold based scaling it is hard to convert a VM capacity to the application performance.

Alternatively researchers proposed set of techniques for dynamic VM reconfigurations. Padala et al. [108] applied control theory based technique. Authors use proportional controller to perform CPU allocation of VM running web application. However, it is difficult to apply proposed technique to control multiple VM resources (such CPU and RAM). One has to identify system model that captures relationship between multiple control inputs and system outputs. To overcome the problem Rao et al. [118] proposed to exploit reinforcement learning. The key characteristic of RL is ability to dynamically learn environment and make decisions under uncertainty based on the environmental observations. The behavior of applications running in a cloud can be affected due to modifications or change in the workload request model. RL based model can detect the changes and catch up desired application resource demand. However, due to large state-space RL requires substantial time to learn and adapt to environmental changes. To deal with problem, RL based approaches either reduce number of observable environment states [118] or apply offline learning [116]. But state-space reduction leads to coarse granular resource allocation.

In this chapter we present VScaler controller. The core of VScaler is reinforcement learning. Our controller performs fine-granular allocation of individual VM resources to meet user provided performance goal. We overcome one of problems of RL approach by applying parallel learning with assumption. It allows to speed up the learning process. Moreover, in comparison to other RL based approaches VScaler does not require offline learning. It dynamically obtains scaling policy.

5.2 MOTIVATION

Most of cloud providers offer virtual resources as fixed size VM and use 1 hour billing period to charge users. However, not every application uses maximum VM capacity during execution

time and runs for a fixed amount hours. Usually average utilization of user VM is below its capacity limit. Moreover, user can launch a VM for 20 minutes to keep up with increased workload and then shutdown the VM. For many users existing pricing model is far from ideal, because it leads to resource wastage. However, the situation is changing. There are IaaS providers such as *CloudSigma* [33] that offers infrastructure with 5 minute billing cycle. Moreover, user is free to construct own VM type by selecting required amount of RAM, CPU or I/O. Flexible bundles model enables cost-efficient VM reconfiguration. For example a user wants to run interactive application inside flexible VM. For the first 10 minutes VM needs 1 GB RAM. After 10 minutes workload increases and the application requires 2 GB of RAM to provide same performance on changed workload. Therefore user adds 1 GB RAM. In the fixed bundles model, to provide same performance user would need to assign upfront VM with 2 GB of RAM. And then pay for 1 GB of RAM which was not utilized during the first 10 minutes.

The example presented above shows that resource usage of the application is not static. It dynamically changes. Many web applications running in a cloud environment such as social networks, online-shops, auctions have fluctuating workload. They belong to the class of interactive applications. The demand of the applications is driven by requests rate of client accesses and characteristics of requests. For example, workload may change from being CPU intensive to memory intensive. It means that scaling policy have to treat individual VM resources. To efficiently provision interactive applications we have to properly design scaling policy, which captures relationship between workload, individual VM resources (such CPU and RAM) and application performance.

Resource provisioning techniques can be classified into two main categories: threshold based and model based. In threshold based technique each application capacity changes based on user defined lower and upper bounds. If resource demand of the application crosses the lower bound then scaling in action is triggered. The scaling out action triggers if the resource utilization goes above the upper bound. The technique is simple and easy to understand. But it fails for control application performance under frequently changing workload. Performance violations occur when the application resource demand crosses the upper bound and resources are wasted if the demand below lower limit. Moreover, there is no way to keep application performance within user-specified constrains.

Model based techniques adapt to workload variation and perform fine granular application performance control. In chapter 3 we present auto-scaling system analysis and an overview of applied model based techniques. Generally the techniques use control theory, queuing theory, time series analysis and machine learning. The first two techniques require domain and application knowledge to build optimal model. Moreover, the model has to be designed offline. It means that any application and workload changes that were not evaluated in the design phase would require rebuilding the model. To dynamically adapt to the changes we can apply two last techniques. However, the time series approach can by applied for workload or resource usage prediction. It does not provide mechanisms to build complex model that can map workload, resource demand and application performance. Alternatively, to design model online we can use machine learning techniques. One of them is reinforcement learning. In contrast to other techniques RL does not require *a priory knowledge*. It is able to perform online model learning and adapt to environmental changes. RL has been successfully applied for dynamic resource allocation [118, 116, 20, 46]. Most of the works apply Q-learning algorithm to explore state-action space. Q-learning is one of commonly used RL algorithms. However, RL has well know problem: state-action space explosion that affects learning time. Therefore number of states and actions is limited. Usually action set consists of three variables: *add*, *do nothing*, *remove* and fixed amount of resources x is assigned at each step of Q-algorithm. For example, VM memory changes by the value of 256 MB [116]. The number of states is defined by the number resource assignment values x that can fit in maximum VM capacity.

Our motivation is to build autonomic VM reconfiguration controller that takes advantage of RL

and can handle fine-granular resource scaling. The controller should perform resource allocation decision based on application resource demand, input workload and user-defined performance goal.

5.3 PARALLEL LEARNING WITH ASSUMPTION

To learn an environment agent in RL takes actions and observes new states. For every action it obtains reward. Therefore the time is takes to activate every action and visit all states of the environment depends on the size of state-action space. Consider VM reconfiguration problem. If we have a VM with n configurable parameters (CPU and RAM), assuming k different settings for each parameter and m actions available from each state. Then minimum number of interactions required to observe rewards for all state-action pairs will be $n^k * m$. Hence, if we want to control two VM resources (CPU and RAM) and each resource has 10 different setting and each state available 3 actions (*add*, *nop*, *remove*). Then total number of iterations required to activate each state-action pair will be $2^{10} * 3 = 3072$.

Statically fixing action is simple. However, it may be not efficient to scale. Consider, following motivating example. VM is assigned CPU cap value 20. Suddenly VM workload increases and it needs more resources. We add cap value of 10. It means that capacity of the VM increases by 50%. But going from cap value 80 to 90 increases capacity by 12.5%. The effect of adding fixed-size resource is not constant. Therefore, using static action values may not be appropriate. Dutta et al. [47] analyzed workloads of cloud based datacenter and found that most of workloads require scaling actions below factor of 2. It means that we need to allow to change VM capacity from any state at maximum by 100%. Assume that minimal action step is 10%. Then we have 10 actions for each state and in total number iterations will be $2^{10} * 10 = 10240$. In comparison to fixed-size resource allocation the number of iterations increased more than 3 times. We need to reduce learning time.

One of the approaches to speed up agent's learning process of approximated model is to learn in parallel [86]. The idea is to run multiple RL agents. Each agent learns by working on individual task and periodically shares own observations with other agents. It means that agent does not need to visit every state-action pair in a given environment. Instead the agent can take the Q-value of state-action pairs that it did not visit from other agents. The approach allows greatly reduce environment approximation time. Parallel learning in RL has been already applied to cloud resource management. In previous work [20] this technique is used to implement autonomic horizontal scaling. The authors defined the environment as a number of VMs divided in to groups. Each group is a observable environment for one agent. Agents periodically share their observations with each other. The work results show that this technique allows significantly speed up the learning process. But approach presented in the paper cannot be simply transferred to vertical scaling. Because it requires run more than one VM to parallelize learning process. Consequently, it increases the cost.

Our solution is based on the idea that there is more to learn from a single transition. Every time when action is taken the agent observes amount of resources consumed by the application and obtains reward for the taken action. However, if there are states where VM capacity is higher than observed resource demand, then we can update transitions that connect initial state with these states.

Consider the following simple example presented on figure 1. Each state represents a assigned VM capacity. First value is memory size in MB and second is virtual CPU capacity expressed in Xen Credit Scheduler [37] cap value. In *state 1* the VM has 768 MB of memory and cap value is 40. The agent takes *action 1* and the VM capacity changes by cap value of 15. In *state 2* the VM has 768 MB RAM and CPU cap value equal 15. After a fixed time interval the agent observes application performance and resources utilization over this period. The performance of the application is within user-specified range. Memory utilization is 83%, virtual CPU

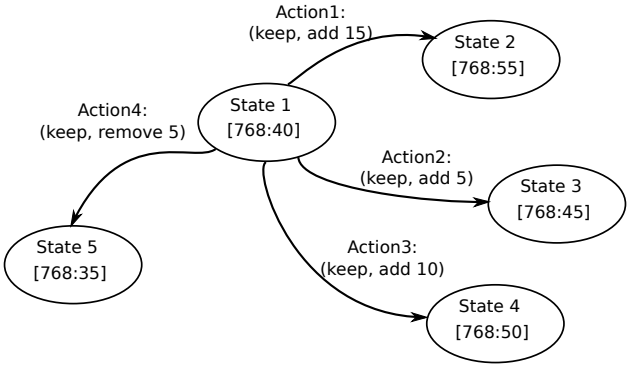


Figure 1: Markov Decision Process with 5 states and 4 actions

utilization is 78%. We could assume, that ideal VM capacity requires $768 * 0.78 = 638MB$ of memory and CPU cap value $55 * 0.78 = 43$. Therefore we can update all transitions connecting *state 1* with states that have RAM and CPU values higher than 638 and 43 respectively. In the presented example we update transitions from *state1* to *state3* and from *state1* to *state4*. The transition from *state1* to *state5* is not updated, because VM virtual CPU capacity in *state5* below 43. Following approach allows to speed up the learning process, because after each agent's action more than one state-action pair is updated.

5.4 VSCALER DESIGN

In this section we provide an overview of VScaler's architecture and work flow. We describe MDP state definition for VM reconfiguration problem, discuss details of each phase of RL and provide description of the algorithm used by VScaler.

We implemented VScaler to perform resource assignment to the interactive application running inside VM. VScaler does not have *a priory knowledge* of the application resource usage behavior. Vscaler adapts scaling policy online. Figure 2 shows overall architecture of VScaler controller and its interactions with external components. VScaler uses a *proxy* monitoring capabilities to get incoming request rate and an application performance feedback. *The host daemon* collects the VM resource usage statistics and implements host's resources allocation to the VM. *The predictor* inside VScaler tracks incoming request rate and predicts workload for the next reconfiguration interval. In order to implement automatic VM capacity management VScaler makes decisions based on the VM resource consumption, the application performance feedback and predicted workload.

The management process runs in a following way. VScaler submits resource allocation scheme to *the host daemon*. *The host daemon* assigns resources to the VM. After the fixed interval of time VScaler requests resource usage statistics from *the host daemon* and the application performance feedback from *the proxy*. The data is used to calculate reward and update capacity management policy of RL model. Then VScaler takes workload prediction and feeds RL model to calculate the best resource allocation scheme for the next reconfiguration interval.

State description

To apply RL to VM reconfiguration problem we have to create state-action space definition. A model of the environment and interactions with the environment in RL described as Markov Decision Process (MDP). We defined MDP for VM reconfiguration problem as $S = \langle m, c, w, g \rangle$, where:

- $m \in \mathbb{N}$ is memory in MB allocated to the VM;
- $c \in \mathbb{N}$ is CPU allocated to the VM, expressed in a Xen Credit Scheduler [37] cap value;

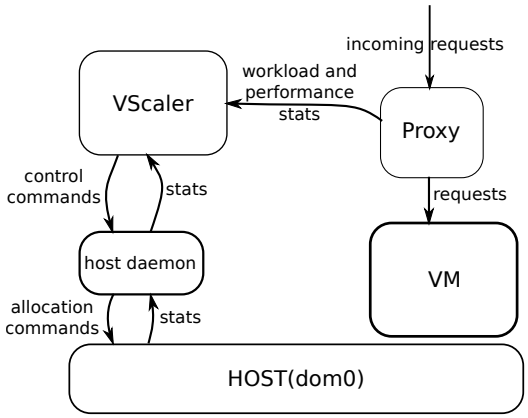


Figure 2: Architecture of VScaler

- $w \in \mathbb{N}$ is a total number of user requests observed per time period and which was served within SLA. This value changes between time steps.
- $g \in \mathbb{N}$ is guess about total number of user requests which can be served in this state without violatinf user-defined SLA. The value is updated using *update alternatives* algorithm.

The agent's action space consists of all allowed actions within current state. The agent can choose to add, remove or keep the CPU and memory allocation. For each resource we assign an action set, which is $A = \{a \in \mathbb{Z}, A_{min} < a < A_{max}\}$. Actions are discretized by setting step a_{st} on each resource. In our experiments memory allocation is bounded between $A_{min} = 512MB$ and $A_{max} = 1536MB$, for CPU we have $A_{min} = 10$ and $A_{max} = 50$.

VScaler uses workload predictor which takes request history as input and applies AR model to predict expected number of requests for the next reconfiguration interval. We do not need to define all possible request rate numbers for each state as it is implemented in [20, 46]. Therefore state-action space size in VScaler depends only on allocation step size for CPU and memory. The last two variables (w and g) do not affect state space size. This design solution reduces agent's environment size and Q-learning lookup table size.

Reward calculation

We use application performance feedback and VM resource usage statistics to calculate reward. Reward function is defined as ratio between $perfFeedback$ and $resUtil$. It guides the agent towards the state that has enough resource capacity to keep the application performance within user-defined range and gives higher utilization of the VM capacity:

$$reward = \frac{perfFeedback}{resUtil} \tag{1}$$

$$perfFeedback = \begin{cases} 1, & \text{if } respTime < SLA \\ e^{-p|\frac{respTime-SLA}{SLA}|} - 1, & \text{otherwise} \end{cases} \tag{2}$$

$$resUtil = \frac{\sum_{i=1}^{n}(1 - U_i)}{n} \tag{3}$$

$resUtil$ is a resource usage efficiency, where U_i is a utilization status of each resource. We consider two resources: CPU and memory. With increase of resource usage $resUtil$ value decreases, it allows to encourage the agent to take actions which give higher resource utilization. We also include SLA penalty in reward calculation. The penalty prevents situations where the application performance degrades, because the agent moves to the states with lowest over-provisioning. To achieve this we set the reward as negative value when SLA is violated.

```
1: repeat
2:     s_t ← getCurrentState()
3:     a_t ← chooseNextAction(s_t, Q)
4:     U_{t+1} ← getResourceUsage()
5:     respTime_{t+1} ← getAppPerformance()
6:     w_{t+1} ← getObservedRequests()
7:     r_{t+1} ← calculateReward(U_{t+1}, respTime_{t+1})
8:     updateModel(s_t, a_t, r_{t+1}, w_{t+1} Q)
9:     if respTime_{t+1} < SLA then
10:        updateAlternatives(s_t, a_t, w_{t+1}, Q, U_{t+1})
11:    end if
12:    updateRequests(s_{t+1}, w_{t+1})
13:    t ← t + 1
14: until Agent is terminated
```

Algorithm 3: Agent learning algorithm

Initializing Q learning

Q-learning is a model-free RL algorithm, where agent learns an environment online. In order to apply control operations during the learning process one has to follow some policy from which decision will be chosen and resource management operations will be taken on controlled system. Defining such policy is complicated, because it requires some knowledge about the application resource usage behavior. In cloud computing context such information may not be available, when an application is deployed for the first time. Therefore only standard policy can be applied. According to [47] we assume that for the next reconfiguration interval the application resource demand can double. Hence, during initialization phase VScaler assigns VM capacity as double amount of currently utilized resources. It allows to avoid application performance degradation during initialization phase. Such approach leads to over-provisioning, but from other side we can update alternative state-action pairs using parallel learning with assumption. VScaler starts to exploit obtained policy, as soon as predictor is ready to forecast workload. In VScaler we use 100 samples of recent observed user requests number to predict the workload for the next reconfiguration interval.

Model learning and exploitation

The agent learning algorithm presented in figure 3. Each reconfiguration interval agent obtains current state then chooses next action. The next action is selected by algorithm presented on figure 4. During initialization phase the agent selects action that increases VM capacity two times in comparison observed utilization from previous reconfiguration interval. If initial policy already obtained, then the agent uses predicted workload value to select next action. To select the action the algorithm takes the state that has guessed requests number g higher than predicted value. Then using list of selected states function *getBestAction* finds transition that have highest Q-value and returns corresponding action. Then the agent takes selected action and observes reward that calculated based on monitored resource usage and application performance. The agent uses reward to update the model. Next achieved performance is analyzed. If SLA was not violated, then the algorithm updates alternative transitions and guessed requests number of alternative states. Finally, at the end of each iteration observed requests number w for the state s_{t+1} is overwritten by w_{t+1}, if $w < w_{t+1}$. However, If SLA was violated then g is overwritten by w, because guess was wrong and amount of resources allocated in state s_{t+1} are not enough to serve observer requests number without violating SLA.

Environment exploration

It is known that RL agents cooperate with managed environment by applying two types of

```
1: if initPhase then
2:     action = DoubleResources()
3: else
4:     predValue ← predictWorkload()
5:     for each state s_next connected to s_t do
6:         g ←getRequests(s_next)
7:         if g > predValue then
8:             selectedStates.append(s_next)
9:         end if
10:    end for
11:    action = getBestAction(s_t, selectedStates)
12: end if
13: return action
```

Algorithm 4: Choose next action

interaction: exploration and exploitation. Exploitation is to follow optimal policy, while exploration is the selection random actions to capture system dynamics and refine the existing policy. Q-learning algorithm uses ε - greedy policy to select an action, where agent makes random action selection with a probability ε. Applying RL in cloud computing context creates additional requirement to the exploration process. One has to ensure that exploration action does not hurt application performance. To prevent performance degradation during exploration phase VScaler selects actions that allocate enough resources to serve predicted workload and then among these actions chooses the one which was executed less frequently. In all our experiments VScaler implements exploration with a probability $\varepsilon = 0.05$

5.5 EVALUATION

In the evaluation we want to answer following questions. First, what is the impact of parallel learning with assumption on the learning speed? Second, how does VScaler perform resource assignment under dynamic workload? To answer these questions we divided the evaluation in two parts. Each part addresses corresponding question.

For the evaluation we build a test bed. The testbed for our experiments is hosted on quad-core Xeon 2.66GHz with 16 GB memory, 100 Mbps network and Ubuntu 12.04 running Xen 4.1. We use RUBiS [121] benchmark to evaluate VScaler. RUBiS is widely adopted interactive application benchmark. In the experiments PHP version of the benchmark is applied. It consists of web front-end and database back-end. We run Apache 2.2 as a web server and MySQL as a database server. The web-server and the database run inside VMs with 64-bit CentOS 6.3. Throughout all experiments only web server is scaled. The database VM is over-provisioned. We present multi-tier application provisioning in chapter 6.

5.5.1 CONVERGENCE SPEEDUP

To show learning speed-up provided by VScaler we evaluated two Q-learning algorithms. The first algorithm uses parallel learning with assumption. The second one is a standard Q-learning approach, where the agent after each observation updates only one state-action pair. To make clear comparison we used the same state-action formalism in parallel learning with assumption as for standard Q-learning approach. Therefore we exclude prediction mechanism. Instead each state was assigned fixed workload value. We also did not use special initialization policy.

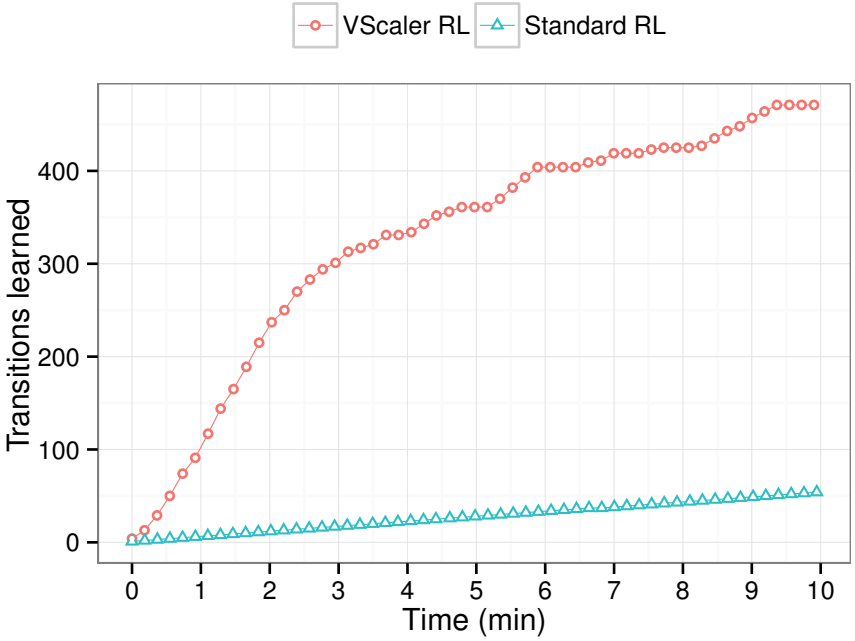


Figure 3: Transitions learned

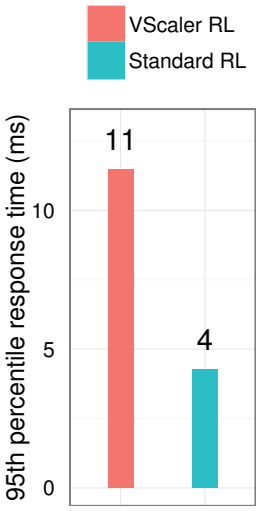


Figure 4: Application 95% response time

In each approach the agent learns the environment using a standard policy. It means that with a probability of $\varepsilon = 0.05$ the agent takes a random action. In the exploitation phase the agent takes an action that gives a higher utility. In the experiment we run a constant workload. 1000 clients were emulated using the RUBiS benchmark. Reconfiguration is performed every 10 seconds.

In figure3 we show the number of transitions learned. During the first 3 minutes VScaler RL learns 300 transitions, while Standard RL only $2 * 60/10 = 12$ transitions. VScaler RL has a higher learning rate, but the most important part is the quality of resource allocation policy obtained by each approach. In figure 4 presented response time delivered by the web server under the control of the evaluated learning models. We ran experiment for 40 minutes. For the experiment we set desired response time to be below 20 ms. Both learning approaches keep 95% of the response time below 20 ms.

The standard RL model has only information about actually visited transitions, while VScaler RL in addition to visited transitions knows about the impact of alternative transitions. The additional information allows VScaler RL to quickly converge to the state with minimal amount of required resources. In figure 5 we present the cost of resources in each state. We assume that in a flexible bundles resource model that allows to dynamically modify individual resource assigned to the VM. For the experiment we took prices from CloudSigma. The figure shows that VScaler RL needs 3 minutes to find the optimal VM size.

VScaler RL quickly adapts to the workload, while Standard RL needs more time to learn the environment. One has to notice that both approaches do not violate the SLA, but VScaler RL in comparison to Standard RL achieves the performance goal for the lower cost.

5.5.2 REAL WORLD SCENARIO

To evaluate VScaler performance in real cloud environment with dynamic resource demands variations we instrumented RUBiS client emulator to modulate request rate of RUBiS benchmark. The RUBiS client emulator reads clients request rate from trace file. The trace consists

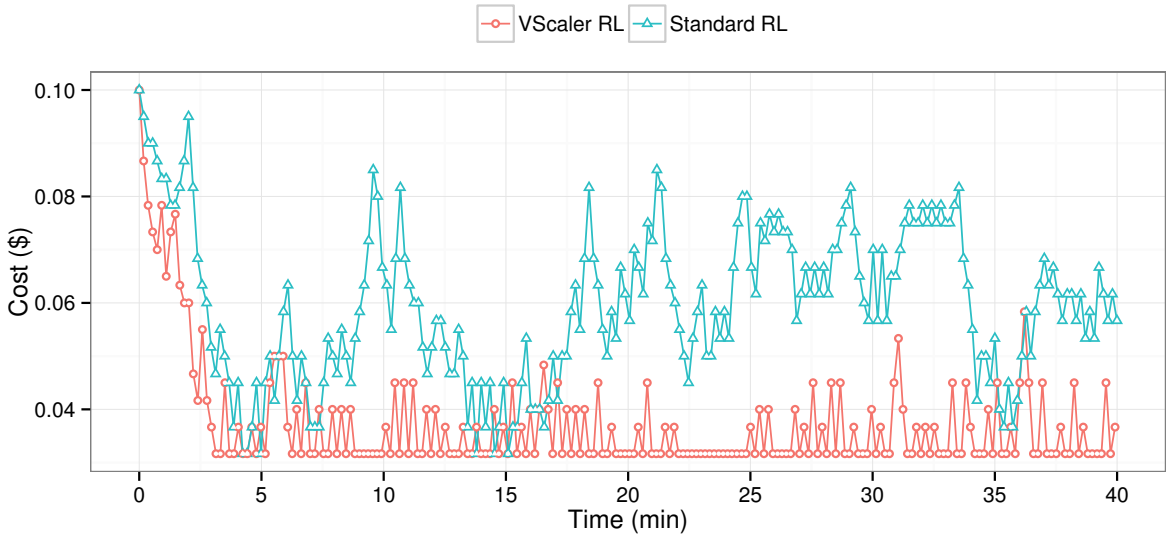


Figure 5: Average costs: Standard RL vs VScaler RL. The greater size of VM in terms of CPU and memory, the greater the cost

of per-minute workload intensity observed during WorldCup 98 [134]. We used 6 hour trace starting at 1998-05-10:03.00.

The goal of this experiment is to measure performance and resource utilization between peak and dynamic resource allocation schemes. Below we present description for each of the schemes. The peak one represents fixed bundle model. We configured VM template for the peak allocation scheme with 1536 MB of memory and 1 virtual CPU with cap value of 50. The template represents Amazon EC2 m1.small instance. To evaluate dynamic schemes we run VScaler in 4 different configurations. In the first two configurations we use fixed step allocation. From each state of the model only 3 action is allowed (*add*, *nop* and *remove*). Hence, CPU and RAM of the VM can be modified only by fixed value. For fixed step allocation we defined small and big step. The other two configurations modify resource with respect to current state capacity. We consider dynamic allocation scheme, because it is not trivial task to find right step size for resource allocation, when an application resource demand changes dynamically. Too big step size leads to over-provisioning, while small step causes resource saturation and leads to SLA violation. In our experiment we set SLA to 20 ms.

- peak allocation
- dynamic allocation
  - fixed step allocation
    - small step (CPU step 2, memory step 64 MB)
    - big step (CPU step 5, memory step 128 MB)
  - flexible step allocation
    - scale up 100%, scale down 100%
    - scale up 100%, scale down 20%

Figures 6 and 7 present the total amount CPU and RAM allocated to web server VM for the workload trace. Peak allocation has highest CPU and memory allocation. It demonstrates how many resources are wasted if web application runs in fixed instance type. The dynamic scheme configurations have significantly lower CPU and memory allocation. Configurations with fixed step allocate slightly less resources than configurations with flexible step. The reason

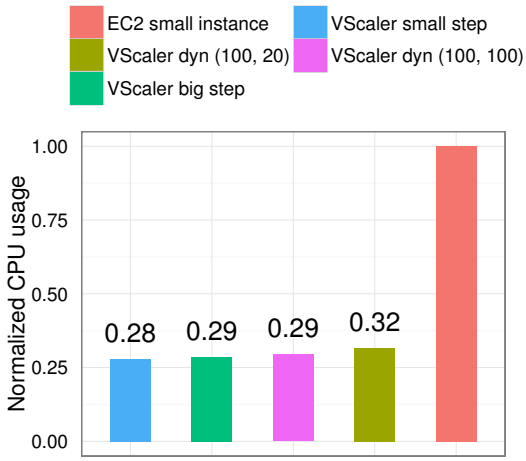


Figure 6: Amount of allocated CPU power

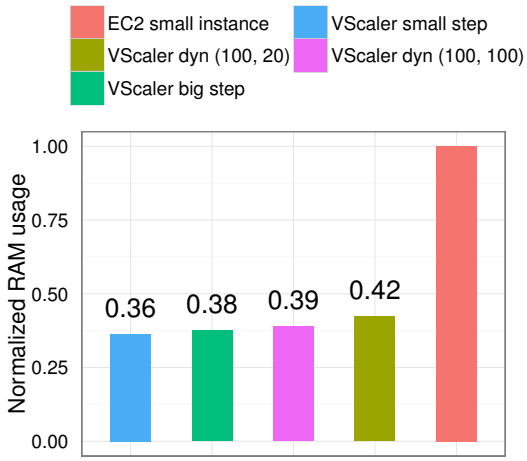


Figure 7: Amount of allocated memory

is that flexible step allocation has more options in scaling. Flexible step configuration with smaller scaling down factor has largest amount of allocated resources among dynamic scheme configuration. It is more conservative on releasing resources.

Figure 8 shows 95% response time. Two configurations violate SLA requirements. As we can see fixed step scaling with smaller CPU allocation size violated SLA, while scheme with bigger CPU allocation step satisfied SLA. The result shows that it is difficult to find 'right' step size when there is no knowledge about application running inside the VM. Another interesting observation is that one of dynamic step allocation schemes also violates SLA. Scheme with 100% scaling down performs aggressive capacity management. In exploration phase it can sharply drop VM capacity by removing a half of assigned resource. However, if after reconfiguration workload increases, then the application cannot provide desired response time. To improve quality of the model one can limit scale down action as we did for other configuration with flexible step. Alternatively, aggressive de-allocation actions during exploration phase can be prohibited.

The evaluation results show that fixed step allocation achieves good performance, but only if allocation step has a proper size. However, one needs to find 'right' one. Configuration with flexible step allocation can dynamically decide how many resources to assign.

5.6 RELATED WORK

Cloud computing industry moves towards pure pay-as-you-go model. More and more IaaS providers shift from fixed bundles to flexible bundles. It means that users have freedom to choose what amount of each resource to allocate. Moreover, minute and second range billing cycles allow to dynamically resize VM, without waiting for the end of an hour. However, one needs to find optimal scaling policy for vertical resource scaling that would perform dynamic resource allocation. A number of attempts have been made to automate the process of VM capacity management. Most of previous work applies control theoretical approaches, machine learning techniques and time series analysis.

A lot of work on dynamic resource provisioning applies control theory. Heo et al. [68] propose controller for CPU and memory allocation. The controller tracks web application response time and adjust VM capacity to keep the application performance within specified range. [109] build auto-scaling system that uses MIMO controller to perform resource scaling of two tier web applications. The controller adapts CPU and disk I/O to achieve user-specified performance

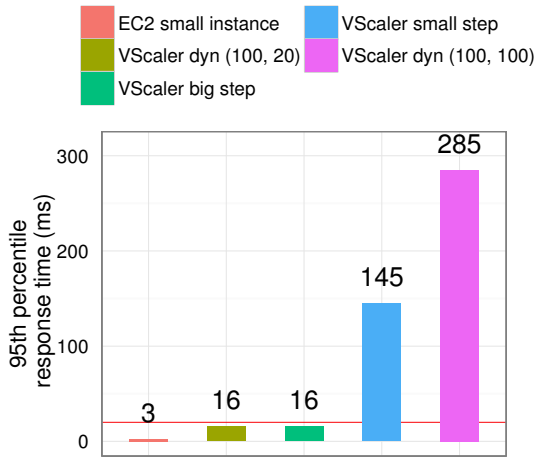


Figure 8: 95th percentile response time

goal. Systems that use control theoretical approaches can provide good performance. However, the design of the system is a complex process. It requires significant time and effort to obtain optimal model.

Rao et al. [116] also apply RL in VM resource management. Authors perform resource allocation of multi-tier web-application. The system presented in the paper uses fixed step scaling. Our evaluation shows that performance of such models greatly depend on the size of the step. Authors do not provide details about the choice of action size. In our work we show that better results can be achieved with dynamic step allocation.

The work by Shen et al. [126] presents CloudScale system. The system uses time series analysis to perform resource usage predictions. In particular authors analyze CPU and memory usage traces to discover repeating pattern and predict future resource demand. CloudScale implements fine granular CPU and memory allocation. However, only one individual resource is modified on evaluated applications. Presented approach is hard to extend for multiple resource management. In contrast, VScaler scales both resources of VM.

SmartScale [47] uses a combination of horizontal and vertical scaling. In each reconfiguration phase authors propose to consider trade-off between these two types of scaling. SmartScale chooses the one that allows better utilize resources and fits SLA requirements. The framework implements reconfiguration with 1 hour period. The chosen period fits to billing cycles of most of cloud providers. But SmartScale would require model adaptation in order to achieve high resource usage efficiency in new conditions. For example, if IaaS providers shrink billing periods.

5.7 CONCLUSION

Cloud computing allows dynamic and fine-granular capacity management of virtual resources. Designing auto-scaling auto-scaling is one of the challenges users face. Reinforcement learning is promising approach in the direction of autonomic capacity management. It allows adapt to dynamic changes of application resource demand. However, adaptation is not an instant process. It requires significant learning time to find optimal scaling policy.

In this chapter we present novel approach to RL that speed ups the learning process and design VScaler controller. The results show that parallel learning with assumption can quickly find optimal resource allocation policy. Moreover, proposed approach keeps application performance within specified service level objectives. We evaluated VScaler against real world

scenario. The evaluation shows that VScaler can efficiently perform VM capacity management.

In this chapter we present single VM scaling that can be applied to a number of applications running in one VM. To provide multi-tier application resource assignment we need to increase state-action space. As we was mentioned size of state-space increases learning time and size of look up time. In next chapter we present RL based approach that reduces complexity RL model.

AUTONOMIC MULTI-TIER APPLICATION SCALING*

*The contents of this chapter first appeared at CLOUD'14 [152].

6.1 INTRODUCTION

It is common today to use public or private clouds. Many users prefer to rent virtual machines instead of using private infrastructure. Typically cloud users allocate predefined VM templates and pay for time the VM have been running. Often VMs run underutilized. Resource demand of many applications is rarely static. It varies over the time. In order to deal with changing load users can allocate resources according to peak demand. However, the peak load provisioning leads to under-utilization and users pay not only for used, but also for wasted resources. Hence, it is desirable for users to have an opportunity to re-size a VM on-the-fly to meet actual resource demand and pay only for resource that have been consumed.

Economic interests of cloud users already affect pricing models of some cloud providers. For example, *CloudSigma* [33] allow to specify desired VM template and change it later during the runtime. Moreover, the provider has 5 minutes billing cycle. It means that a user can modify VM every 5 minutes. We see that pricing model offered by *CloudSigma* makes possible cost-effective scaling. Users can dynamically acquire and release individual VM resources to provision own applications. But what is the appropriate policy to control resource allocation without affecting the application performance?

Traditionally resource allocation process addresses only the application resource demand. Most of cloud providers assign VM capacity based on resource usage threshold. However, for cloud user the most important objective is performance of the application hosted on the cloud. Especially, if the application belongs to the class of interactive applications. For e-commerce website low latency is crucial requirement. No one wants to interact with slow responding application. Hence, scaling policy should provide resource assignment based user-defined application performance goal. Moreover, many web applications are multi-tier component systems. Resource provisioning of one tier does not necessary lead to overall application performance improvement. In [116] authors show that changing capacity of one tier can lead to utilization increase of another tier. The complexity of task increases if we can tune individual VM resources. Therefore, scaling policy model needs to address cluster wide correlation effects.

Resource scaling policy design is a complex process. It requires running controlled experiments. Existing data collected from production systems is hard apply for the application performance modeling. It often lacks sufficient information about all relevant correlations between input-outputs of the system. Moreover, due to hardware heterogeneity the model obtained for the same type of VM can vary greatly [157]. Hence, we need a technique that can learn and adapt scaling policy online.

Researchers propose techniques to enable online policy learning. The techniques can be divided in two groups. The first group [159, 140] applies so called 'sand-box' approach. However, it requires setup of 'sand-box' that requires specific implementation for each application. Moreover, for the 'sand-box' cloud operators have to assign dedicated hardware. The second group [133, 118, 117] uses reinforcement learning (RL) approach. It is model free technique that does not require *a priori* knowledge about the application and virtual environment. It learns the application resource usage behavior online. However, in section 3.4.2 we mentioned that RL based approaches suffer from what is known as the curse of dimensionality: an exponential explosion in the total number of states as a function of the number of state variables.

In this chapter, we describe our self-adaptable resource scaling controller, called VscalerLight. It automatically generates the required scaling actions and triggers them to guarantee SLA requirements. The core of VscalerLight is RL approach. Our design is based on analysis of a web application behavior under different resource allocation configurations. With the help of the analysis we propose to split memory and CPU controller models instead of tightening them together. The use of individual controller per each resource reduces the state-space complexity and eliminates well-known problem of RL based controllers. To orchestrate resource allocation across all application tiers, we add workload parameter to each tier model. VscalerLight does

not require offline initialization. Alternatively, it uses knowledge base exploration.

6.2 MOTIVATION

Dynamic application scaling is non-trivial task. There are number of challenges to address. The following questions arise during the process of the scaling policy implementation.

First, conversion of SLA to resource allocation. User that deploys an application in cloud environment expects certain performance from the application. To control the application performance the user can change capacity of a VM. However, It is difficult to determine the *'right'* amount of CPU and RAM that needs to be allocated to achieve desired performance. Hence, we need to obtain correlation between the resources allocated to the VM and the application performance.

Second, time-varying resource demand. Many web applications have highly fluctuating workloads. It means that resource demand of the application also varies of the time. Static resource allocation for these applications can lead to either over-provisioning or under-provisioning. Both cases are not desirable. If an application provisioned according to the average load, then performance of the application degrades. From another side peak load provisioning leads to resource wastage. The application does not utilize allocated capacity, because peaks load are rare. To deal with resource usage fluctuations we have modify VM capacity with respect to current demand. Therefore prediction mechanism is required to anticipate the fluctuations. Some of the demands can be predicted. For example, if they have daily, monthly or seasonal patterns. However, there are cases when it is hard to provide high prediction accuracy. Unexpected raise of popularity of a website cannot be predicted. In such case we need to perform reactive scaling. Therefore dynamic scaling policy should predict future demand and perform reactive scaling if unexpected load spike occurs.

Third, multi-tier applications scaling. Multi-tier applications require appropriate resource allocation across all tiers to provide performance specified in SLA. Changing capacity of one tier can lead to the shift of load to another tier [116]. Therefore it is necessary to create a model that captures relationships between individual tiers of the application.

Forth, dependencies between individual resources. Application needs to access multiple system-level resources to provide desired performance. Hence, it is necessary to perform multi-resource provisioning.

Cloud provider does not have not knowledge about the user's application. For example, if the application is deployed for the first time. It makes difficult to provide correct resource allocation policy for the application. Therefore most of cloud providers offer easy and lightweight auto-scaling service based on thresholds. The idea of the approach is to assign or release certain resource according to user-predefined threshold. For example, when CPU utilization reaches $ThUp = 60\%$ a new VM is allocated and the VM is deallocated if CPU utilization drops below $ThDown = 30\%$. The service user has to set these thresholds. It means that the user has to have expertise knowledge or evaluate the application offline in order to define the 'right' thresholds. Moreover, as we mentioned above, the ultimate goal of resource allocation is to guarantee application performance under different workload conditions. Threshold based scaling does not provide mechanisms that can be used to specify application performance goal. Alternatively, cloud provider could offer auto-scaling service that takes as input application quality of service requirements and generates scaling policy online.

The problem described above inspires us to look for an approach that allows to adapt scaling policy during runtime. One of common approaches is applying adaptive control [82, 53, 83, 108]. However, to use adaptive control we need to obtain the model of the system and the environment dynamics. In contrast, reinforcement learning(RL) can find optimal system model online. RL generates scaling decisions from observation of the system during runtime. However, there is a still need to address some challenges, such state-space complexity. We aim to

design controller that performs vertical scaling of multi-tier application. It means that each VM has at least 2 configurable parameters (CPU and RAM). To achieve fine-granular allocation the number of available values for each of the parameters should be large enough. But the increase of the parameters values can make use of RL approach impractical. The size state-action will affect learning time. Therefore we have to simplify RL based model. In the next section we describe how the complexity of the model can be reduced.

6.3 SYSTEM IDENTIFICATION

Designing the formal system model is complicated and time consuming process. It has to be repeated each time when the workload pattern changed or and application is updated. The goal of our identification experiments is to understand how different control knobs affect the application performance. During application lifetime the quantitative relationships between resource capacity available for the application and its performance can change due to workload dynamics or application updates. However, the fundamental impact of each resource to the application performance remains the same. We aim to design controller that controls CPU and RAM assignment to VM running components of interactive application. Therefore we run set of experiments to understand the impact of each of control knobs on application performance and its components (tiers).

For the experiments we created a Xen-based test-bed that consists of representative multi-tier web application benchmark - RUBiS [121] (PHP version). RUBiS is a free, open source auction site prototype which simulates real users' behavior of a popular auction *eBay.com*. The front-end tier is a Apache http web-server (WS), the back-end is MySQL database (DB). Each tier runs on VM with centOS6. Client requests are issued by dedicated group of machines running RUBiS client emulator. With certain probability clients access different pages of the application. The clients can browse or bid. Browsing does not utilize database tier, while bidding causes significant load on DB. We run mixed workload that has mixed types client accesses. The experiments consist of two parts.

6.3.1 CPU USAGE AND PERFORMANCE

In first part we analyze impact of virtual CPU power on the application response time and resource demand. Therefore we periodically changed the CPU entitlement and monitor application performance. We use Xen credit scheduler [37] to assign virtual CPU capacity to the VM. CPU allocation cap value varied from 8 to 100. To understand the effect of workload fluctuation we also modified number of clients sending requests. The number of clients changes from 400 to 1600 by step 400. CPU allocation experiment was conducted for each VM running the application tiers.

Figure 1 shows the mean response time (MRT) as function of front-end tier CPU entitlement. If we increase CPU cap value, then for each request rate there is a maximum CPU entitlement value that affects MRT. Above the value MRT does not change any more. The graph 2 gives alternative view on CPU and response time correlation. It represents relationship between CPU utilization and MRT. Each data point is an average from 20 samples. The presented measurements show that changing CPU entitlement provides smooth control of MRT when CPU utilization above 80%. The same correlation we found between DB CPU entitlement and the application response time. Lower utilization levels do not allow to control CPU. It means that we need to keep CPU utilization above certain threshold to control the application response time.

The change of CPU entitlement also affects web server memory consumption. It is shown in figure 3. Memory utilization increases with increase of CPU utilization. The reason is that higher CPU utilization leads to increase of MRT (see figure 2). Therefore incoming requests instead

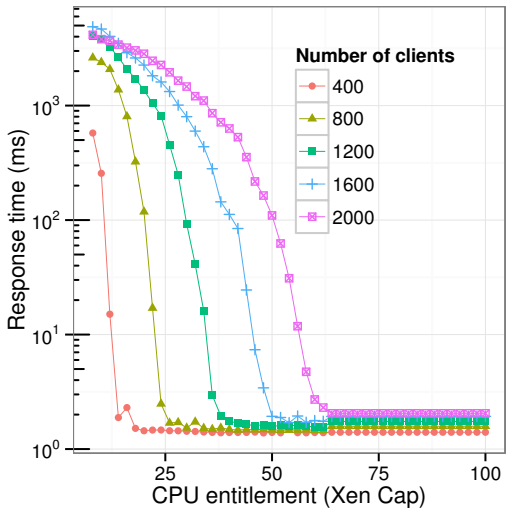


Figure 1: MRT vs CPU entitlement

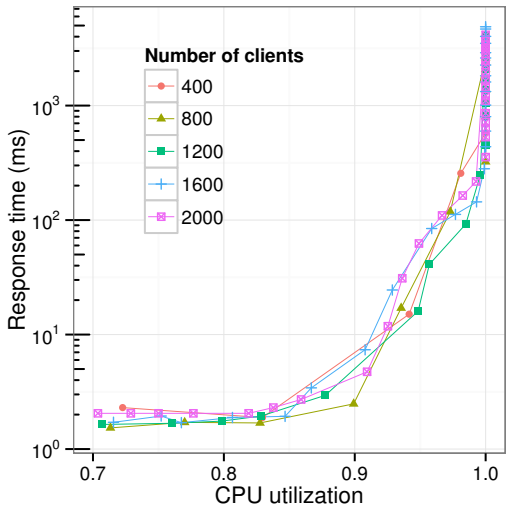


Figure 2: MRT vs CPU utilization

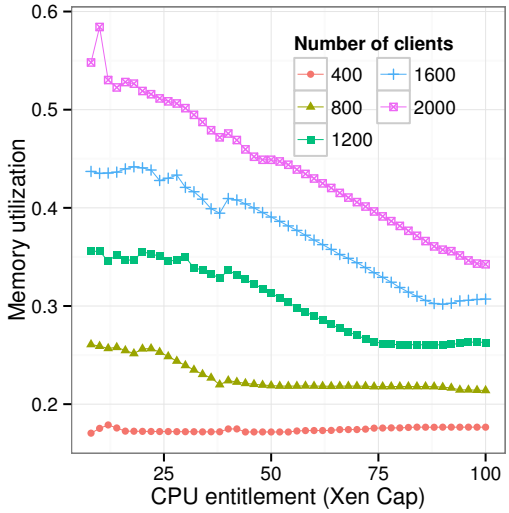


Figure 3: WS memory utilization vs CPU entitlement

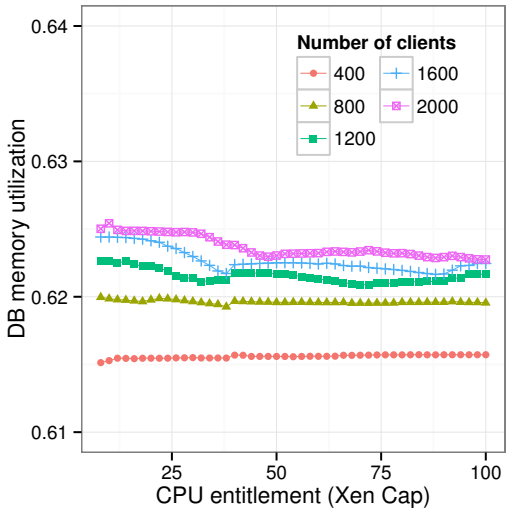


Figure 4: DB memory usage vs WS CPU entitlement

of being served immediately go to the application queue. As a result the queue size increases and it needs additional memory space. The same effect has web server CPU entitlement on database memory usage (see figure 4). Web server cannot process responses of DB tier and DB has to keep them in memory. Reduction the power WS VCPU leads to decrease of DB CPU usage. Figure 5 shows the effect. Basically low power web server CPU needs more time to process incoming requests.

6.3.2 MEMORY USAGE AND PERFORMANCE

In the second group of experiments we evaluated the effect of VM memory capacity on the application performance. The memory allocation varied from 240 MB to 896 MB. The experiment was repeated for different workload intensity of 400, 800, 1200, 1600 clients. We limit minimal VM memory capacity to 240 MB. This is a minimal RAM size required by underlying OS. We cannot go below the value, even if actual memory usage is lower.

Figure 6 presents the relationship between response time and memory utilization. MRT sharply increases when memory utilization reaches 90% threshold. Memory pressure (ratio

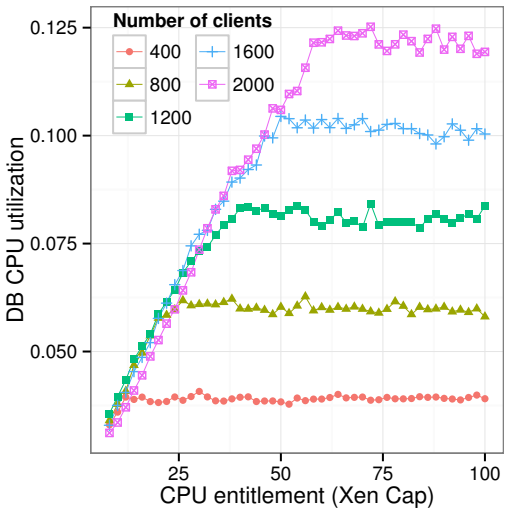


Figure 5: DB CPU utilization vs WS CPU entitlement

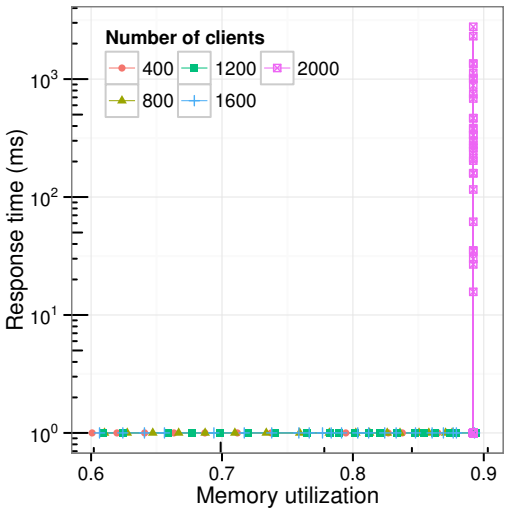


Figure 6: MRT vs memory utilization

between VM capacity and actual memory consumption) does not affect response time below the threshold. The reason is swapping activity that is shown in figure 7. During swapping process OS tries to free up memory by saving memory pages to a disk. The speed of the disk orders of magnitude slower than memory, therefore the application performance is affected dramatically. We conclude that memory cannot be used for smooth MRT control. Memory is *non-compressible* resource that cannot be reclaimed without severely affecting application performance. In some case it can lead OOM (out of memory) events, when OS start to kill tasks to free up memory.

6.3.3 CLUSTER WIDE CORRELATION

Resource provisioning of multi-tier applications should provide fair resource allocation across all tiers to avoid shift of resource bottlenecks. Therefore it is important to understand how the change of resource allocation on one of tiers affects the resource consumption of another tier. Based on our previous experiments we analyze cluster wide correlation effects. In figure 8 presented the correlation between DB CPU entitlement and WS memory utilization. The graph shows that the memory usage increases if the CPU entitlement is reduced. Higher request rate leads to higher memory utilization levels. The reason is that request service rate μ provided by available CPU capacity of DB tier is less than incoming request rate λ. DB cannot accept connections from WS and WS stores incoming requests in the queue instead of sending them down to the DB tier. In figures 4 and 5 we show that different web server CPU entitlement changes memory and CPU usage of database tier. The knowledge of cluster wide correlation is important when one of the tiers runs under resource pressure. In such situations the second tier is over-provisioned. If scaling policy does not consider this effect, then it would add more resources to the first tier and reclaim resources from the second tier. As a result it brings the second tier to saturated state and the first tier to over-provisioned state. Finally, the policy would not be able to solve instability problem. To avoid described scenario we can use simple approach: increase allocation of all tiers if one of them is close to saturation. After the application performance stabilizes go back to normal scaling policy.

The following can be concluded from the experiments results. CPU and RAM belong to different groups of resources. CPU is *compressible* resource (as well as network and disk I/O bandwidth) and it can be reclaimed from the application by decreasing it performance. In contrast memory is *non-compressible* resource (as well as disk space) and we cannot reclaim memory if it is already used by the application. Only free VM memory can be reclaimed. Hence,

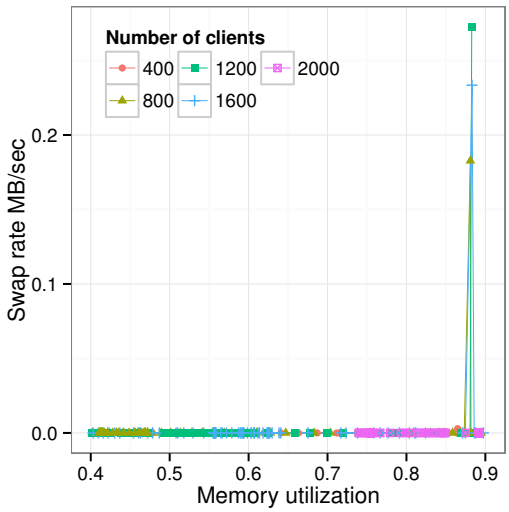


Figure 7: Swap rate vs memory utilization

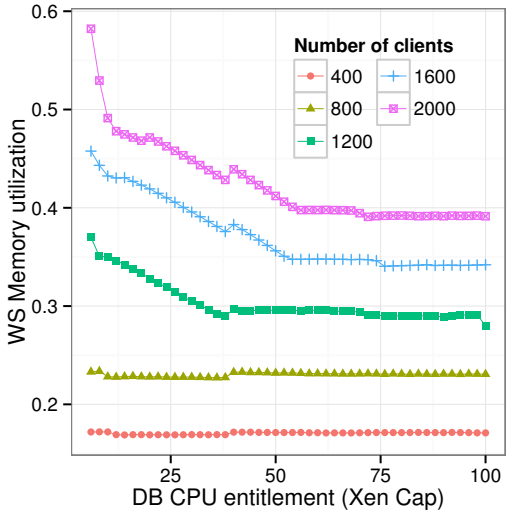


Figure 8: Effect DB CPU entitlement on WS memory usage

it is important to keep memory utilization below the value that triggers swapping process. To regulate the application response time we need to change CPU entitlement.

6.4 CONTROLLER ARCHITECTURE

6.4.1 OVERVIEW

We designed and implemented online resource scaling controller for multi-tier application. The controller does not require *a priory* knowledge of the application performance model. With the help of reinforcement learning approach VscalerLight learns scaling policy online. Our controller has predictive and reactive mechanisms. Reactive mechanism allows to quickly scale up the VM resource assignment in response to unexpected load spikes, while predictive component assigns resources in advance.

Our controller runs on top of Xen hypervisor. It consists of five main components: monitor, predictor, CPU module, memory module and capacity manager. The monitor collects the application performance and resource usage statistics. VscalerLight has dedicated CPU and memory modules for each VM. Modules contain RL models and output the resource allocation scheme. The capacity manager performs resource allocations by communicating with underlying hyper-visor. Predictor tracks incoming request rate and issues the value of the workload for the next reconfiguration interval.

On figure 9 presented the implementation of VscalerLight. The resource management is organized in a following way. The monitor tracks the application performance and resource consumption metrics U_{CPU}, U_{RAM}. The load balancer (LB) presented in the picture provides the monitor with performance metrics. Resource consumption is collected via API provided by the hypervisor. The predictor forecasts the request rate value for next reconfiguration interval. VscalerLight runs reconfiguration every 10 seconds. Each resource controller takes predicted value and outputs resources entitlement values A_{CPU}, A_{RAM} for each tier. Then capacity manager performs VMs reconfiguration. After the fixed time interval the summary statistics are collected and the models inside CPU and memory modules are updated. If performance of the application violates the value specified in SLA, then VscalerLight immediately recalculates resource allocation values and reconfigures VMs.

VscalerLight prediction module uses auto-regressive (AR) model to anticipate future workload. The prediction value for the next reconfiguration interval is calculated based on previous

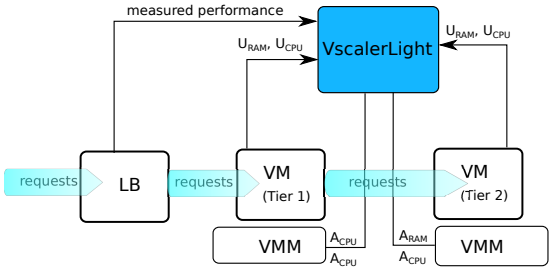


Figure 9: VscalerLight implementation

100 samples. VscalerLight has also an option to react quickly before the end of the reconfiguration interval if the unpredicted load spike occurs and response time crosses value specified in SLA. In this case VscalerLight takes the current request rate value and asks CPU and memory modules for appropriate resources entitlement. If the entitlement found, then VscalerLight performs resource allocation. If the 'right' resource entitlement value is not found, then VscalerLight shifts to the exploration phase.

6.4.2 MDP DESIGN SOLUTIONS

Out experiments in the section 6.3 show that resource consumption has a positive correlation with incoming workload. Both CPU and memory usage increases with the increase of the load. Therefore we have to include workload dynamics component into RL model. We also observe that changing memory entitlement does allow to regulate the application response time. Therefore we can exclude memory from the response time control model. However, we still need to control memory allocation and avoid memory swapping. Therefore we create separate model for memory. The workload parameter value included in each model will orchestrate CPU and memory models.

CPU model state definition We define MDP which models our approach to VM CPU allocation problem as $S = \{(c, w) | 0 \leq c \leq C_{max}\}$, where:

- $c \in \mathbb{N}$ is CPU allocated to the VM; this value is expressed in Xen credit scheduler cap value. $C_{max} = 100$
- $w \in \mathbb{N}$ is observed request rate which was served without violating SLA. This value can change over time. Initially it is set to 0.

Memory model state definition We define memory model state-space as following $S = \{(m, w) | M_{min} \leq m \leq M_{max}\}$, where:

- $m \in \mathbb{N}$ is memory allocated to the VM. $M_{min} = 256$ MB and $M_{max} = 1536$ MB.
- $w \in \mathbb{N}$ is observed request rate which was served without violating SLA. This value can change over time. Initially it is set to 0.

The action set for CPU and memory models is defined as following $A = \{a \in \mathbb{Z} | A_{min} \leq a \leq A_{max}\}$. The actions is bounded between $A_{min} = -50\%$ and $A_{max} = 50\%$. It means that the entitlement values c or m can change within these bounds. The following idea lies behind our actions-space definition. We think that is more efficient to change the VM resource capacity by certain percentage rather than add or remove fixed capacity value. Adding fixed value may be not efficient to scale. For example, a VM is assigned CPU cap value 20 and workload increases. Then VM needs more resources. If we add cap value of 10, then capacity of the VM increases by 50%. But going from cap value 80 to 90 increases capacity by 12.5%. Hence, effect of adding fixed-size resource is not constant. Therefore, using static action values may not be appropriate. In chapter 5 we present evaluation of fixed size and flexible size capacity allocation

policies. The results show performance of fixed size capacity allocation policy greatly depends on resource allocation unit size, while flexible size policy dynamically determines optimal action.

Reward function facilitates the conversion of SLA to VM resource assignment. We use application performance feedback and VM resources usage statistics to calculate the reward. It is designed in such a way that it guides the RL agent towards the state that gives higher utility:

$$reward = \frac{perfFeedback}{resUtil} \tag{1}$$

In section 6.3 we found that CPU provides smooth response time control. Hence, we defined *perfFeedback* as following:

$$\begin{cases} perfFeedback = 1, & if respTime < SLA \\ e^{-p|\frac{respTime-SLA}{SLA}|} - 1, & otherwise \end{cases} \tag{2}$$

The agent gets negative reward if SLA violation happens, otherwise the reward value depends on CPU utilization. Our analysis in section 6.3 states that memory utilization should be bellow $hUtil = 90\%$. We have to make sure that memory capacity under particular workload does not trigger swapping process. Therefore memory model reward depends on the value. It is possible to dynamically determine the threshold. But for simplicity we leave it fixed.

$$\begin{cases} perfFeedback = 1, & if memUtil < hUtil \\ e^{-p|\frac{memUtil-hUtil}{hUtil}|} - 1, & otherwise \end{cases} \tag{3}$$

The resource utilization of each model is defined as following.

$$resUtil = \frac{(1 - U_r)}{n} \tag{4}$$

where, U_r is observed resource utilization from the last reconfiguration interval.

The controller performs resource allocation across all tiers. As we found in section 6.3 resource modules for each tier can work independently as soon as there is no SLA violation. If the violation occurs, then it is hard to determine 'right' control knob that can bring the system to a stable state. Therefore in case of SLA violation we shift all resource modules from exploitation to initialization phase.

6.4.3 INITIALIZING Q-LEARNING

To perform the environment exploration we apply Q-learning algorithm. Q-learning is a model free RL algorithm. The scaling policy is learned by taking actions and observing a system feedback. Initially there is no policy available. To initialize RL model we can either learn based on statistical data [116] or apply guided exploration [24]. Our assumption is that no statistical data is available upfront. Hence, we apply knowledge based exploration. The idea is simply keep resource utilization within desired bounds. For CPU and memory these bounds are 50% and 80%. During the exploration phase the agent takes actions that keep the resource utilization inside the interval. The initialization runs until the predictor collects enough data to perform prediction.

6.4.4 MODEL LEARNING AND EXPLOITATION

Initially the *w* parameter of the state description of each model is set to 0. This value changes during the exploration phase. The main loop of Q-learning algorithm is presented on figure 5. The algorithm is the same for both CPU and memory models. Each reconfiguration interval the agent chooses an action from the policy learned so far and after fixed interval of time collects

```
repeat
    s_t ← getCurrentState()
    a_t ← chooseNextAction(s_t, Q)
    U_{t+1} ← getResourceUsage()
    r_{t+1} ← calculateReward(U_{t+1})
    w_{t+1} ← getObservedRequests()
    updateRequests(s_{t+1}, w_{t+1})
    updateModel(s_t, a_t, r_{t+1}, w_{t+1} Q)
    t ← t + 1
until Agent is terminated
```

Algorithm 5: Agent learning algorithm

```
predValue ← predictWorkload()
for each state s_next connected to s_t do
    g ←getRequests(s_next)
    if g > predValue then
        selectedStates.append(s_next)
    end if
end for
return getBestAction(s_t, selectedStates)
```

Algorithm 6: Choose next action

resource usage and the application performance statistics. Then it updates states that satisfy observed load and finally updates the policy.

CPU model state update After the action a_t has been taken w is updated by w_{t+1} for each state where CPU entitlement value c provides lower or equal CPU utilization than observed utilization U_{t+1} . This update rule guarantees that all updated states can serve request rate w with the same MRT. See figure 2.

Memory model state update The state request rate value w is updated by w_{t+1} if memory entitlement value m in the state is higher than observed memory consumption.

The exploitation phase algorithm is presented in figure 6. The algorithm takes workload prediction value and selects states that have w higher than predicted value and connected with current state s_t. Then from the list of selected states it takes the one that has higher Q-value and gets action that moves the agent to the selected state.

6.5 EVALUATION

In our evaluation we compare VscalerLight against threshold scaling policy and two static allocation schemes. Cloud user can use auto-scaling service offered by provider to perform dynamic resource assignment. However, it is necessary to find optimal threshold values for the application. Therefore users often use default thresholds $ThDown$ = 30% and $ThUp$ = 60%. These thresholds minimize the probability of under-provisioning. Capacity reclaimed when utilization is fairly low and allocated when there is a still large enough room for additional load. However, such policy can lead to high resource wastage. For our evaluation we created two additional policies. First policy tries to minimize level of over-provisioning. The thresholds for policy are $ThDown$ = 70% - $ThUp$ = 80%. The second one aims to minimize SLA violation events. The

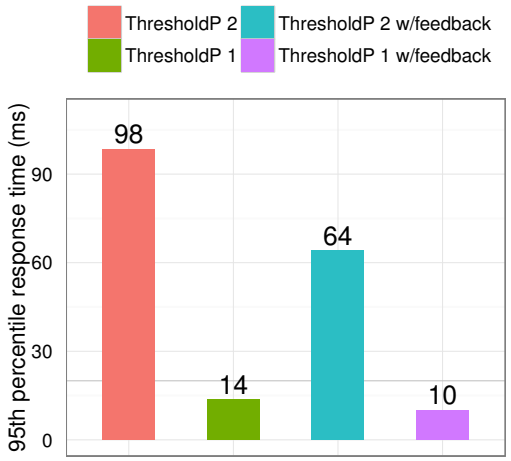


Figure 10: Threshold based policy: with and w/o performance feedback

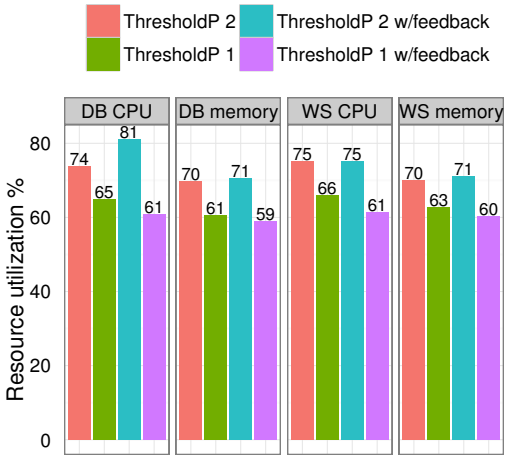


Figure 11: Threshold based policy: average resource utilization

thresholds for policy are *ThDown* = 50% - *ThUp* = 80%. We evaluate only vertical scaling, therefore we defined scaling unit for each policy in terms of memory and CPU entitlements values assigned to a VM. Usually threshold polices operate with VMs. Policy 1 is allowed to add CPU entitlement by value of 2 and memory entitlement by value of 8 MB. The second policy can add 5 and 16 respectively. Resource de-allocation unit for both policies is fixed. We don't allow to reduce VM resource capacity more than 10%. For static allocation schemes we assume, that a user knows expected resource demand and assigns VM resources according peak demands. For the first scheme user takes fixed size VM (1536 MB, CPU cap 60), which is equal to Amazon EC2 small instance [11]. For the second one user specifies VM resources.

For the evaluation we use the test bed presented in section 6.3. The testbed for our experiments is hosted on quad-core Xeon 2.66GHz with 16 GB memory, 100 Mbps network and Ubuntu 12.04 running on top of Xen 4.1. VscalerLight is evaluated against real world scenario. We apply workload trace from the World Cup 98 [134]. It consists of HTTP requests made during the 1998 World Cup Web site. For our experiments we use 6 hour trace starting at 1998-05-10:03:00. The trace has high fluctuations. The ratio between min and peak loads is 12. We define desired application response time to be no greater than 20 milliseconds.

Threshold based policy does not have application performance feedback, while VscalerLight obtains application performance metric. To improve it we extend threshold based policies with performance feedback. Whenever the application performance crosses target response time each policy allocates additional capacity. In figure 10 we plot response time provided by the threshold policies. Figure shows that adding performance feedback reduces the application response time. Cumulative distribution line of both policies moves to the left after adding performance feedback. Moreover, performance feedback improves resource usage. Figure 11 presents resource utilization for each tier. CPU utilization of database tier under control of the policy 2 increases by 7%. If the policy is aware of the application performance, then it does not trigger allocation action even if the threshold is crossed. Performance feedback reduces false positive allocations. In contrast, CPU utilization of DB under control of the policy 1 decreases. The policy lower bound threshold 20% higher and it less sensitive than the policy 2. After we added performance feedback it triggers allocation earlier, because it is aware that response time of the application already affected.

In figure 12 we show the application response time and resource assignment trace over 8 minutes. CPU allocation of web server and database is expressed in Xen cap values. The policy 1 has high level instability. CPU entitlements of DB and WS jump up and down. The

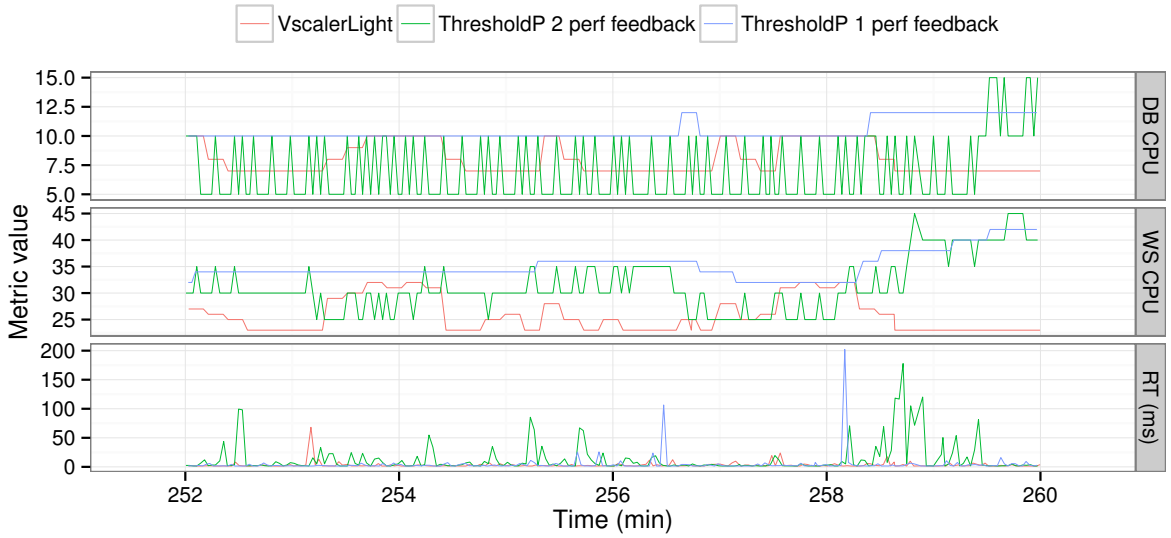


Figure 12: Response time

DB VM mostly affected. The policy 2 provides better control. It does not create fluctuations. However, in comparison to VscalerLight it allocates more resources. Moreover, response time trace shown in the bottom of the graph shows that VscalerLight and the policy 2 provides similar performance.

Figures 13 and 14 show the performance and resource usage provided by evaluated schemes. The policy 2 violates SLA, while the rest schemes keep the application response time below specified value (20 ms). If we look on figure 14, then see the efficiency of each allocation scheme. The higher the value, the better resource usage efficiency. Fixed size VM allocation leads high resource wastage. Resource utilization of both VMs is below 26%. It means that in cloud market user overpays for about 74% of allocated resources. If user can customize VM resources, then VMs utilization can be improved by factor of 2 in comparison to fixed size VM. However, one needs to know workload upfront. Dynamic resource scaling improves resource utilization even further. In all dynamic schemes the utilization above 59%. However, only two of them (VscalerLight and policy 1) meet user specified performance objective. But VscalerLight achieves 10% higher utilization in comparison with threshold policy 1 and does not require tuning of the thresholds upfront.

6.6 RELATED WORK

There are many works in the direction of dynamic resource allocation that address the problem multi-tier application scaling. They use a wide range of techniques. Researchers apply control theory approach [108, 68, 83], queuing models [136, 158], time series analysis [126, 56], machine learning [118, 47, 88].

Padala et al. [109] design MIMO adaptive controller. The controller adjusts CPU and disk I/O of multi-tier application VMs to meet user defined performance objectives. Presented controller can adapt to different operating regimes and workload conditions. Kalyvianaki, Charalambous, and Hand [83] also implemented MIMO controller for multi-tier web applications. Authors integrate Kalman filter into feedback controller to dynamically allocate CPU resources to the application VMs. Kalman filter is well known technique to remove noise from data. Presented approach can follows workload change without creating transition fluctuations. The controllers implemented in the paper can dynamically adapt to workload changes. However, to obtain the

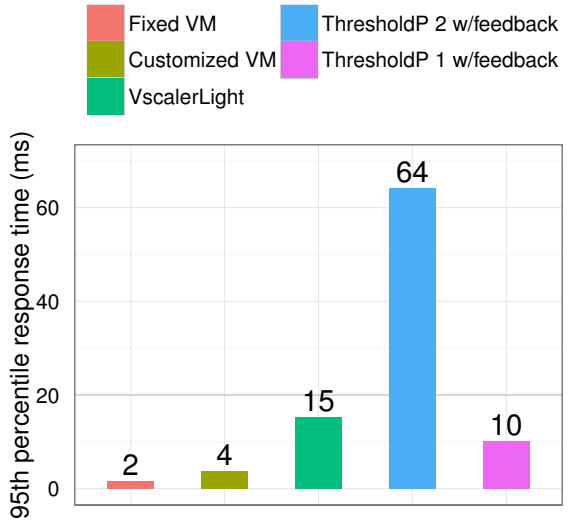


Figure 13: 95% response time

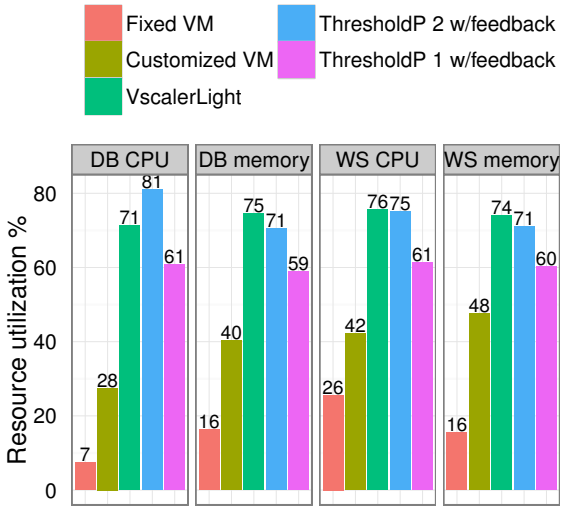


Figure 14: Average resource utilization

application performance model one needs to run offline system identification process. In our work we designed online self-adaptive controller that does not require initial application performance model.

Urgaonkar et al. [136] and Zhang, Cherkasova, and Smirni [158] use the classical queuing theory to model application multi-tier application. The queuing theory is used for the system with stationary nature. Therefore in case of scenarios with changing conditions the parameters of the queuing model have to be reconfigured. In the presented paper authors addressed the problem of scaling multi-tier application using queuing network. The provisioning of the application is performed based on the peak load. As we show in evaluation peak load provisioning leads to high resource under-utilization. Hence, a user overpays for resources that have not been consumed.

Shen et al. [126] proposed a framework which analyzes repeating patterns in resource usage traces to automatically scale application VM capacity. The framework performs CPU and memory scaling, but it scales application tier in isolation from other tiers. In the evaluation authors do not apply the framework to scale multiple tiers of web application. Moreover, pattern matching algorithm has number of parameters that need to be tuned. In the paper these parameters were obtained empirically.

In [88] authors present machine learning based techniques to model the performance of VM-hosted applications. In particular they apply neural networks and support vector machines. Applied techniques trained on collected application performance data. The techniques demonstrate low rate of prediction errors. But, one needs to obtain the performance data. Authors assume 'sand-box' approach where user can deploy the application and run sample workload.

Our work is closely related to RL based resource allocation approaches. Rao et al. [118] implemented VM capacity management system to perform resource assignment of multi-tier application. The core of the system is Q-learning algorithm that is used to discover the application performance model. Authors include each tier VM parameters into a single RL model. Applied RL model design increases state-space size. In order to improve learning time authors use two approaches. First, they reduce resource scaling granularity. Second, they apply neural network (NN) approximation. However, NN brings training overhead. Authors report that the training takes about 10 minutes and it has to be periodically repeated. In our work we designed small per resource model that has N-size state-space complexity. It enables us to assign resources with small granularity.

6.7 CONCLUSION

Cloud resource selling model shifts towards flexible bundles model. In contrast to fixed size bundles users can specify amount of resources they need. Moreover, VMs can be reconfigured during runtime. It means that the user can follow the application resource demand by performing fine-granular VM scaling. However, without understanding the application performance model and workload dynamics it is difficult to make proper scaling decisions.

In this chapter we presented VscalerLight, autonomic resource scaling controller for multi-tier applications. VscalerLight does not require *a priory* knowledge. It dynamically learns application performance model. The core of the controller is RL approach. In contrast to existing RL based systems, we use simplified RL model. Hence, we are not limited on resource allocation granularity. Presented evaluation results show that VscalerLight efficiently allocates resources across all tiers of the web application and meets user-defined performance objective.

VM reconfiguration together with small billing cycles improves virtual resource usage. However, the maximal size of VM is limited by capacity of host machine. There is need to enable horizontal scaling for workloads which resource demand is beyond the host capacity. For the future we plan to extend VscalerLight to provide combination of vertical and horizontal scaling.

I/O AWARE ELASTIC MAPREDUCE CLUSTER SCALING*

*It is an extended version of the paper that first appeared at CLOUD'15 [155].

7.1 INTRODUCTION

Today timely and cost-effective analytics on 'big data' are a key requirements for business and science. Web search engines and social networks store in logs users activity information. Periodically the logs are analyzed to find user behavior patterns and provide personalized advertising or detect suspicious actions. There is also growing demand for data processing from scientific fields such as biology, astronomy and economics.

Large-scale data processing frameworks are the tool to work with "big data". Running these frameworks on a private infrastructure requires high upfront expenses and also leads to complementary maintenance costs. Therefore there is increasing interest in running such data processing frameworks in cloud environment. Cloud provides access to unlimited amount of virtual resources that average user can rent and pay for allocated resources with respect to well-known pay-as-you-go model.

To simplify the process of cluster provisioning large cloud providers such as Amazon and Microsoft deliver data processing services that exploit MapReduce computational model [40]. For example, Amazon offers *Elastic MapReduce* (EMR) [12] a Hadoop based web service for "Big data" data processing. A user can run variety of jobs such as web indexing, data mining, log file analysis, machine learning, scientific simulation, and data warehousing. In contrast to Hadoop, where each node of the cluster works as data storage and compute unit, *EMR* has dedicated storage and compute nodes that divide the cluster into data and compute parts. The separation enables elasticity properties which is hard to provide in traditional Hadoop architecture. The node containing data cannot be simply removed from the cluster. The removal of the data node either leads to data loss or degradation of fault tolerance properties in case of replicated data. But the compute part scaling does not cause such effects. A compute node can be easily added or removed to/from the cluster to speed up computation or improve utilization of the cluster. However, the architecture of EMR cluster creates traffic between data and compute nodes that scales together with the compute part [87]. It raises a cluster sizing problem, because the network throughput and the data part capacity remains fixed.

Cloud users can scale number of VMs dedicated to the application, but for the best of our knowledge none of public IaaS providers scales network capacity together with increased number of user's VMs. Hence, if the network cannot sustain increased traffic, then it becomes a bottleneck. There is also growing interest to run data processing jobs across multiple clouds [48, 73]. Iordache et al. [73] proposed framework that allows to run MapReduce applications across number of clouds. Cross-cloud computation requires to transfer data over WAN that has even weaker performance properties [148] in comparison to inter datacenter networks.

The data nodes of EMR store and serve persistent data which imposes additional constraints on elasticity properties of the data storage. Adding a new data node does not give immediate performance improvements to the storage, because the nodes do not have any persistent data. The new node must wait until data rebalancing procedure is complete. Hence, there is no incentive to resize the data storage at job runtime.

In presence of aforementioned bottlenecks the incorrect choice of the compute part size can significantly stretch the task completion time of MapReduce job. It increases the total task completion time that impacts the final resource rental cost in the cloud. Pay-as-you-go model and elastic nature of the platform allows the user to change the size of data processing cluster almost instantaneously. Agmon Ben-Yehuda et al. [4] observe that IaaS providers shift from hour range to seconds range billing cycles. It allows users to remove exceeding virtual resources at any point of time, without waiting for the end of billing cycle [89, 26]. However, to achieve cost efficiency the user has to scale compute nodes with respect to the data part capacity and available network throughput. Average user does not have knowledge about performance delivered by these resources. Moreover, each MapReduce job has own data traffic model. Hence, there is a need for system that can resize compute cluster of EMR to minimize the cost

of job execution in the cloud.

In this chapter we propose online resource scaling technique that allows to find appropriate cluster size for each MapReduce job. The presented approach does not require multiple runs of a job, which is common for cluster sizing techniques [149, 69]. ElasticYARN determines the size from the single wave of a job execution and adapts number of containers running in parallel with respect to bandwidth provided by the data storage and the network.

7.2 MOTIVATION

The workload of data processing clusters dynamically changes due to size of data that needs to be processed or job type [119]. During day hours the load increases and drops in the evening. Some jobs are CPU intensive, others I/O intensive. The fluctuation of the workload results in different cluster resource usage. Therefore, running fixed size cluster is not cost-efficient. If the cluster runs in the cloud environment then user is going to overpay for resources that are not used all the time. To take advantage of pay-as-you-go model and adapt to workload variations we need to scale data processing cluster.

Traditionally [40] nodes of data processing cluster works as data storage and computing unit. The goal of such design is providing data locality. Accessing data from local hard disk gives higher performance in comparison to remote execution, since there is no need to transfer data over network. However, such cluster cannot be easily scaled in and out. One can add more nodes, but removing of exceeding computing power is not trivial. Each of the nodes contains data and removing it from the cluster means that we lose part of the data. To avoid this we have to move the data to the remaining nodes of the cluster. However, there are two obstacles to consider. First, the remaining nodes should have enough capacity to store the data. Second, the overhead of data transfer can be large. Every time when we resize the cluster we would need to transfer the data back and forth. Therefore, it is better to decouple the data from the computation. The resulting cluster consists of two types of nodes: data nodes and compute nodes. The data nodes compose data storage and have limited scalability. We can add and remove data nodes, but it requires to trigger load-balancing that may require substantial time to complete [95] before increased the data storage capacity takes an effect. It is more practical to scale almost stateless compute part. Amazon implemented model of separated data and computation in EMR cluster. Users can provision compute cluster to perform processing of data stored in Amazon S3 storage. EMR cluster allows to scale the computing part with respect to resource demand of particular MapReduce job. From another side it creates significant traffic between data and compute nodes that scales with the compute part.

Over last few years a number of cluster management frameworks such Mesos [71], YARN [141], Quasar [41] were developed. The frameworks perform resource management of data-processing job to ensure resource allocation fairness among tasks running in the cluster. Each task of the job runs inside a container that exploits Linux Groups for resource isolation. Containers can isolate wide range of resources dedicated to a certain task. However, the frameworks perform only CPU and RAM isolation, while leaving I/O resources consumed by the task without attention. Therefore, containers are scheduled based on available cluster capacity in terms of CPU and RAM. EMR uses Hadoop framework to perform resource management. It means that scheduling tasks in EMR cluster would not properly treat traffic between the data storage and the compute nodes.

For example, a job with 64 map tasks arrives to the cluster with available capacity equal to 70 containers. Each map task requires 10 MB/sec bandwidth. Then the scheduler will run all 64 maps in parallel that would require total throughput of 640 MB/sec. However, if the data storage has only 3 nodes equipped with one disk each, then at maximum they can provide 360 MB/sec performance (assume disk throughput 120 MB/sec). In figure 1 we show the impact of increased parallelism (number of containers launched in parallel) on the job completion time and

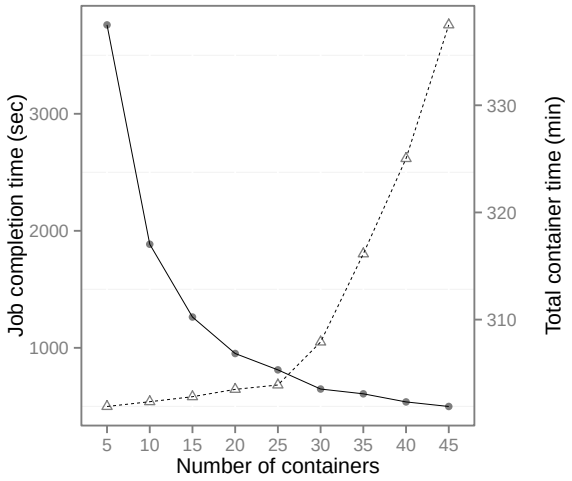


Figure 1: Impact of increased parallelism: Job completion time (solid line) and Total container time(dashed line)

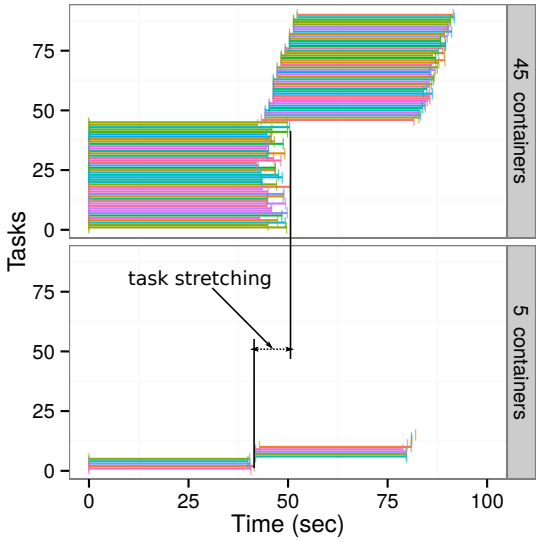


Figure 2: Job execution waves

the total container time. *The total container time is sum of all containers (tasks) runtime*. Tasks running in parallel compose a wave (see figure 2). Allocating larger waves allows to reduce job completion time. If the data storage cannot sustain increased throughput demand, then the total container time increases with the size of the wave as it is shown in figure 2. In presence of the bottleneck the task completion time stretches. Higher total container time means higher cost for cloud user.

However, the non-scalable storage is not only the bottleneck for EMR cluster. For MapReduce applications it is intuitive to increase resource N times to complete a job N times faster. This simple model can be applied to CPU and RAM. However, in case of network it is more complicated [87]. We are not aware of large public cloud providers that scale network bandwidth together with increased number VMs. Hence, if the total bandwidth demand from the compute nodes is higher than throughput provided by the network, then the nodes will end up competing for the network bandwidth. Hence, the total container will increase.

We also observe a trend towards cross-cloud data processing [35, 48, 73] and use of Micro-clouds [131, 98]. The competition between cloud providers gets tougher. Therefore, the prices of cloud providers constantly change. Some of them may offer resources even free of charge. Hence, for the user it would be beneficial to have freedom to move from one cloud to another. In case of large data processing clusters moving all the data is costly and time consuming. But if the cluster has separated the data and the compute parts, then the user can run the compute part on the cheaper cloud. For example, Microsoft offers its own service for 'big data' analytics, called HDInsight [15]. Users are free to choose the data storage either from Microsoft or other provider. It is possible to use existing Amazon S3 service. In case of Micro-clouds it is impossible to deploy the whole data processing cluster in one cloud. Usually Micro-clouds consist only of few servers, for example, 10 or even less, and have restricted resources. Therefore, the cluster has to be distributed across number of Micro-clouds. Running data processing in cross-cloud fashion incurs over WAN data transfer. It is known that WAN has weaker performance properties [148] in comparison to inter data-center networks. Authors of the paper report that the maximum bandwidth between two data-centers located in Illinois and Texas is 465Mb/sec and average latency is $27\ ms$. According to the well-known pay-as-you-go model cloud users pay based on the time they occupy virtual resources (e.g. VMs). If the network is slow then they end up paying for the network traffic, even if the data transfer itself is free.

Existing cloud provider-based data-analytic frameworks such Amazon EMR, Microsoft Azure

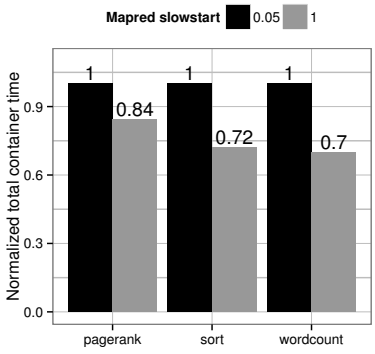


Figure 3: Total container time

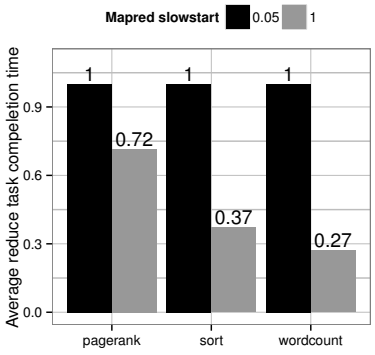


Figure 4: Reduce phase total container time

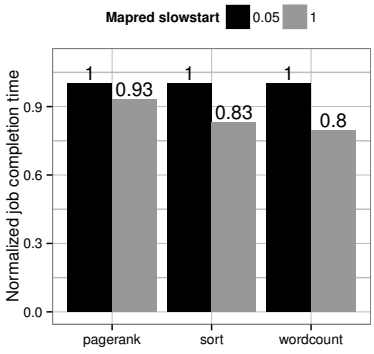


Figure 5: Job completion time

HDInsigh do not scale automatically. Services only provide monitoring tool and control knobs that users can use to make decision about cluster scaling. Moreover, existing resource management frameworks do not include I/O resource into scheduling decisions. Therefore, users have to manually determinate right number of compute nodes. To find optimal cluster size that minimizes the cost of MapReduce job execution users would need to a run each job with different number of containers. For the jobs that appear one time only there is no way to find optimal cluster size with described approach. It motivates us to look for technique that allows to determine optimal number of containers during the job runtime.

7.3 BACKGROUND

ElasticYARN designed and implemented on top of YARN [141] framework. YARN is the second generation of Hadoop platform. Hadoop is one of well-known open source implementations of MapReduce computational model. Hadoop has two types of nodes: the master and the worker. The master node performs job scheduling by assigning tasks to the worker nodes. Every worker node has fixed number of map and reduce slots. The worker nodes periodically report their status and amount of free slots to the master. The master uses the reports to decide where to assign new tasks. All submitted jobs in Hadoop are managed by single master. In large deployments number of worker nodes can grow up to 4000 [103]. Due to high management overhead the master becomes a bottleneck.

YARN was designed to overcome the limitation of initial Hadoop implementation. The design of YARN decouples computational programming model from the resource management. YARN consists of three entities: resource manager (RM), node manager (NM) and application master (AM). In YARN each submitted job has dedicated application master that performs application specific scheduling. The capacity management is done by RM. YARN employs container technology to perform cluster wide resource scheduling. A container is a scheduling unit that isolates memory and virtual CPUs available for the task. For MapReduce applications there are no more reduce and map slots. Each phase task gets a container. AM specifies the number of containers that it needs and sends a request to RM. RM estimates available resources and sends back response, which contains NM location information and the number of containers that AM can launch for the job.

7.3.1 MAPREDUCE SLOWSTART PARAMETER

In the first generation of Hadoop framework each worker node had a fixed number of map and reduce slots. During the map phase all reduce slots were idle and waited for completion

of the map phase. It resulted in low utilization of worker nodes. To improve it the *slowstart* parameter was introduced. The *slowstart* parameter defines a fraction of the number of maps in the job that should be complete before reduces are launched. The start of reduce task earlier reduces a job completion time, because the map phase execution overlaps with the reduce phase. However, in YARN the parameter has different effect.

In YARN each task runs inside a container. There are no special containers for each phase task. Hence, all available containers can be first dedicated to the map tasks to fully utilize available NMs. The overlapping of the map and the reduce phases in YARN also allows to shrink job completion time. However, starting the reduce phase during the map phase execution increases the reduce phase completion time. YARN does not limit the number of reduce containers. Hence, freshly started reduce tasks would occupy all available capacity of the cluster and map tasks that have not been launched yet would have to wait for completion of running tasks. In some cases it may cause deadlocks, because a reduce task can finish only after all map tasks are complete. Starting the reduce phase earlier also leads to increase of the total container time. As a result, the cost of job execution in the cloud increases too.

Default *slowstart* value in YARN is 0.05, so the reduce phase starts as soon as 5% of map tasks are complete. To show the impact of *slowstart* parameter in YARN we run small benchmarks. We set *slowstart* to 0.05 and 1. Figure 3 presents the total container time. The graph gives an estimation of resource usage time. Running MapReduce jobs with *slowstart=1* gives from 15% to 30% cost reduction in comparison with *slowstart=0.05*. In the case of *slowstart=0.05* the reduce phase tasks start earlier, but they run longer, since there is a need to wait for completion of map tasks. Figure 4 shows that reduce task completion time decreases significantly if *slowstart* is set to 1. Moreover, from figure 5 we can see that *slowstart=1* allows to reduce the job completion time. Our primary goal in this work is to reduce total task completion container, therefore for rest of paper we set *slowstart* to 1. It means that the map and the reduce phases do not overlap.

7.3.2 ANATOMY OF MAPREDUCE JOB

A MapReduce job consists of two phases: map and reduce. At the beginning of the job the data is read from data storage and then processed by the map phase. Then the result of the map phase it shuffled across all reduces. Finally, the reduce phase pushes results back to data storage. In this work we aim to optimize the map and the reduce tasks runtime. Therefore, we look into details of each task type execution.

A map task has 5-6 stages: *read*, *compute*, *collect*, *sort*, *spill*, *combine* and *merge*. In *compute* stage, the map task reads data split from the storage and applies a map function to each key-value pair. In *collect* stage it stores output of the map function in the buffer. If there is more data to process, then the map task repeats previous stages. The *sort* stage sorts output key-value pairs before spill occurs. The *spill* starts as soon as data buffer reaches user specified threshold. By spilling map output to a local disk the map task empties the buffer. The execution of the task is blocked during data spill. The *combine* stage is optional, user may not specify it. If the stage is specified, then the map task performs local combine. Finally, the map task executes the *merge* stage, which writes data to local file system. The *combine*, the *merge*, the *spill* stages are disk I/O intensive, while *read* is network I/O intensive.

A reduce task consists of 3 stages:*shuffle*, *merge*, *reduce*. In *shuffle* stage reduce task fetches the map phase outputs from NM. Then in *merge* stage it merges the outputs and writes the result to the local disk. Finally the *reduce* stage invokes the user-defined reduce function and writes the result to the data storage. In the reduce task the *shuffle* and the *reduce* stages are network intensive, while *merge* stage performs communications with the local disk.

Figure 6 shows data flow in EMR cluster. In the map phase data is read from distributed storage (data part). The maximum read throughput is equal to $min(L * D, L_{net})$, where L is read

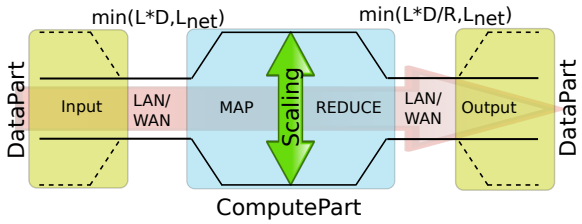


Figure 6: EMR cluster data transfer bottlenecks

performance provided by each of D data nodes and L_{net} is available network throughput between the data and compute nodes. In the reduce phase final output of MapReduce job is written back to the storage. The write performance is determined by $min(L_{net}, L * D/R)$. It is similar to the map phase, excluding R parameter, which is replication factor(usually $R > 1$). Data reads and writes over the network also occur during the *shuffle* stage of the reduce phase. However, the communication model during the *shuffle* stage is different. Amount of data transferred during the *shuffle* stage does not increase with increased number of reduce tasks in the wave [87]. Moreover, in the *shuffle* stage the data is transferred inside local network, which we consider as non-bottleneck resource.

7.3.3 ANATOMY OF LINUX NETWORK STACK

To understand the effects of the data transfer on resource usage we briefly look in to Linux network stack. On the way to destination the data passes four layers of the stack: session (sockets and files), transport (TCP), network (IPv4) and link (Ethernet). Below we describe data flow and control flow of data transmission and receive operations.

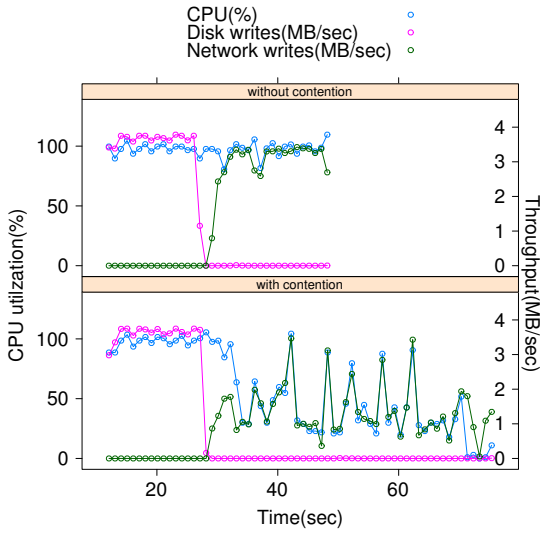


Figure 7: Sort: reduce task resource usage

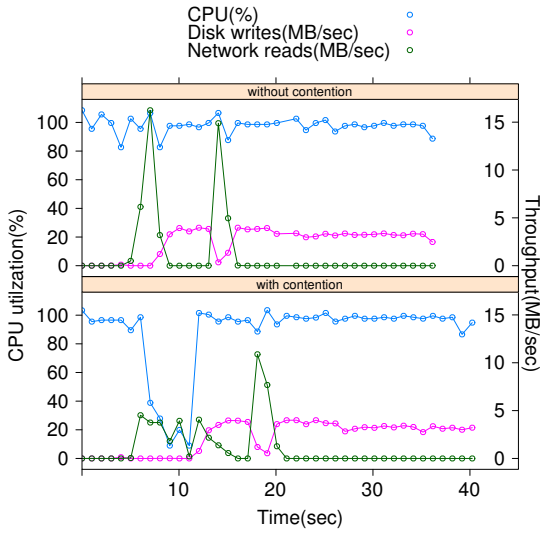


Figure 8: Sort: map task resource usage

DATA TRANSMISSION

When the application wants to transmit the data, it calls *write(fd, buf, len)* function. The required connection socket is identified by the file descriptor - *fd*. POSIX - operating systems expose the socket to the application in the form of file descriptor. After the call of *write* function the data is copied from the user space to the kernel memory and added to the send socket buffer *sk_buff*. The buffer space pre-allocated for each socket. If the buffer runs out of space, then communication stops. The data remains in the user space, until the buffer becomes available

again. In case of non-blocking call *write* returns an error. After the data successfully copied to the kernel, TCP layer is called. If the current TCP state allows data transmission, then a new TCP packet is created. However, if the data transmission is impossible, due to flow control, then the system call stops at this point and control returns to the application. After the TCP packet is created and all IP-routing procedures are performed the device driver requests data transmission. Finally, Network Interface Card (NIC) copies data from the main memory to its memory and sends the packets to the network.

The description of data transmission path shows that the application, which data send rate is higher than throughput provided by underlying layers, will end up waiting for the network layer to become available. The slowdown occurs either because of overflowing buffer in kernel space or TCP flow control mechanisms. From resource usage perspective it means that the application will not use CPU cycles, since the application first needs to send the data to continue execution. Such effect we see in reduce task execution. Figure 7 shows resource usage of the container running sort job's reduce task with and without contention on the path to remote storage. The contention means that the total send rate of reduce tasks in the wave higher than $min(L_{net}, L * D/R)$. If there is no contention during communication with remote storage, then CPU utilization is stable. However, in case of contention the CPU utilization drops. We can see that after the merge stage is complete (disk write activity drops to 0), reduce task starts to send data. The send rate is significantly lower, if we compare against the upper graph. The reduce task running with contention completes in 80 seconds, while the task in the upper graph completes in 50 seconds.

DATA RECEIVING

Data receiving is procedure that handles incoming packets. To receive the data a network device pre-allocates a number of *sk_buff*. The number is configured per device. At the time of a packet arrival NIC generates an interrupt for one of the server's CPUs. Then the CPU executes kernel interrupt (*irq*) handler. The interrupt handler takes the *sk_buff* and processes it further to the network layer. Based on the type of the received packet (ARP or IPv4) it will be handled by different functions. We are interested in IP4v packets. Later on the header of IPv4 packet is parsed and the packet is checked for validity. On successful check it is sent to TCP layer. The processing of incoming packets on TCP layer is done as follows. If user process already waiting for data to arrive, then data will be immediately copied to user space. Otherwise, the *sk_buff* is appended to one of the socket's queues and will be copied later. Finally, at this layer the receive function signals that data is available and wakes up the process. On the session layer the data is copied from the socket or the application needs to wait for data to arrive.

Similar to data transmission path, the low data receive rate (due to network contention or slow send rate from the data part) will block the application execution. Hence, the application that needs to read data is idling. Figure 8 shows two cases of map task execution: without and with contention. In the first case the total read rate of map tasks running in parallel is lower than throughput provided by network and the data part. In the second case it is higher. If we look at the bottom graph, we see that due to contention CPU utilization drops during data receive procedure, while in the upper graph CPU utilization is stable along the task execution. From the figure we see that map task in lower graph at the beginning gets only 5 MB/sec of bandwidth, while it needs about 15 MB/sec. Therefore, it takes about 6 seconds more to complete in comparison to the map task in the upper graph.

7.4 APPROACH

To find optimal number of containers we need to quantify the impact of the bottlenecks on the map and the reduce task completion time. In previous section we show that execution of

MapReduce job's tasks stretches during data transfer between the data and the compute parts. Slow communication leads to low CPU utilization when the task reads and writes data from/to the remote storage. However, CPU usage is not affected during local disk write operations. As we have seen from map and reduce tasks execution models, network and disk write activities do not happen at the same time. Therefore, we divide the runtime of the map and the reduce tasks in two components. First component is the time spent for communication with remote storage, we call it *communication time*. Second component is the time spent on the disk writes, we call it *disk time*. To identify the *communication time* we look at the disk write activity of the task (see figures 7 and 8). The write performance drops almost to 0 during communication with the data storage. The pink line in the figures shows the disk writes during each of the phases.

After *the communication time* is discovered, we need to calculate the maximum number of tasks that we can launch in parallel without stretching them. Assume two cases of tasks execution: with and without contention. In both cases a task needs to process S bytes. However, without contention the task gets B_0 bandwidth and needs T_0 seconds to process the data. In case of contention it gets B_1 bandwidth and takes T_1 seconds to complete. Since the amount the data that needs to be processed in both cases is the same, then:

$$T_0 * B_0 = T_1 * B_1 \tag{1}$$

The total amount of CPU cycles spent by the task during *the communication time* is equal too. Therefore, following condition holds:

$$T_0 * CPU_0 = T_1 * CPU_1 \tag{2}$$

where C_0 and C_1 is CPU utilization with and without resource contention during communication time. If we take equations 1 and 2, then

$$B_0 = B_1 * \frac{CPU_0}{CPU_1} \tag{3}$$

In section 7.3.3 we show that without contention among the tasks running in a wave CPU utilization CPU^{comm} during *the communication time* is equal to CPU utilization CPU^{disk} during disk writes. Therefore in equation 3 we can replace CPU_0 with CPU_1^{disk} and CPU_1 with CPU_1^{comm}. Finally, required bandwidth calculated as follows:

$$B_0 = B_1 * \frac{CPU_1^{comm}}{CPU_1^{disk}} \tag{4}$$

To determine whether the task was running under contention we need to calculate α parameter:

$$\alpha_1 = \frac{CPU_1^{comm}}{CPU_1^{disk}} \tag{5}$$

Value of $\alpha < 1$ indicates that there is resource contention and we need to reduce size of the wave.

Maximum throughput that can be achieved between the data and the compute parts is determined by the minimum throughput delivered by the data storage and network. In the map phase it is $min(L * D, L_{net})$ and in the reduce phase $min(L * D/R, L_{net})$ as is shown in figure 6. Usually the user does not know these values upfront. Alternatively, we can apply following idea. The maximum throughput available when tasks run with and without contention is equal:

$$N_0 * B_0 = N_1 * B_1 \tag{6}$$

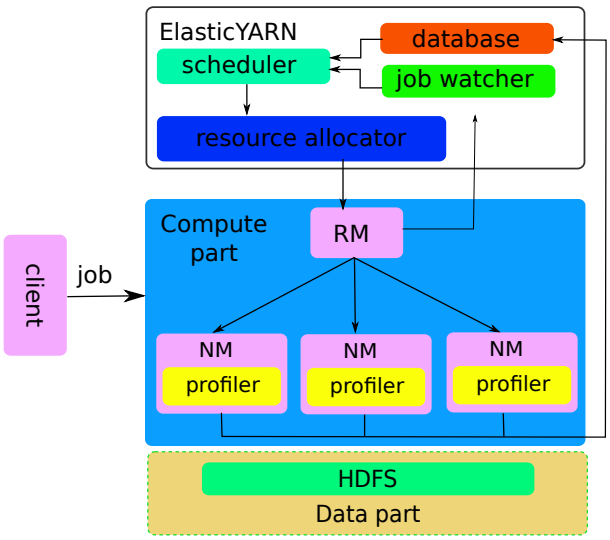


Figure 9: ElasticYARN architecture

where N_0 and N_1 is the number of tasks in each case. Finally, by combining equations 6, 4 and 5 we calculate the maximum number of tasks that we can run without hitting limits of the data part and the network.

$$N_0 = \frac{N_1}{\alpha} \tag{7}$$

7.5 SYSTEM ARCHITECTURE

To demonstrate the effectiveness of presented approach, we have developed a system for EMR, called ElasticYARN that determines optimal number of tasks during the job execution.

7.5.1 OVERVIEW

Figure 9 presents ElasticYARN architecture. ElasticYARN runs on top of YARN and consists of four components: profiler, scheduler, database and job watcher. The profiler runs as a daemon on each NM and monitors network traffic of containers running inside NM. The data collected by the profiler is stored in the database. We use *Redis* key-value store to save jobs profiles. For the interested readers we refer to our previous paper [114], which gives more details about *Redis* and the profiler communication. The scheduler runs algorithm 1 and communicates with YARN to change the number containers running in parallel. The job watcher tracks jobs submission, theirs progress and notifies the scheduler.

7.5.2 JOB PROFILE COLLECTION

To apply our approach we have to collect YARN containers resource usage statics. The task in performed by the profiler. The profiler is implemented as python module, which monitors network traffic and CPU usage of YARN containers running MapReduce tasks. To monitor the traffic we look for communication between NM containers and the nodes that belong to the data part of the cluster. As input the profiler gets list of data nodes IPs and counts sizes of the packets received and transmitted by particular container from/to data nodes. Every second the profiler reports to the database the resource usage statistics.

7.5.3 JOB RESOURCE ALLOCATION

To control bandwidth consumption we have to explicitly specify the number of containers we launch in a wave. In YARN there are two main types of resource schedulers: capacity and fair scheduler. The schedulers are responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. Capacity scheduler is default YARN scheduler. It was designed to allow different departments within an organization to share the cluster. It uses queue to provide capacity guarantees. The queue shares are specified in the form of % of the cluster capacity. Moreover capacity scheduler provides elasticity properties to the queues. The unused capacity of a queue can be harnessed by overloaded queues that have a lot of temporal demand. However, the scheduler does not allow to set the capacity of the queue in the form of exact amount of cluster resources, so that the application can get guaranteed amount of containers.

The second available scheduler is Fair scheduler. The scheduler organizes applications into queues and share fairly resources between these queues. By default, all users share a single queue, named as "default". If an application specifically lists a queue in a container resource request, then the request is submitted to that queue. In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum and maximum shares to queues. The shares are expressed as a number of virtual CPUs and RAM. Using this mechanism we can control number of containers assigned to the application. For example, if a container size is 1024 MB and 1 vcore (virtual CPU) and maximum shares parameter is 4096 MB 6, 4 vcore. Then YARN can run up to 4 containers of an application in a wave.

The algorithm. On the job submission actual limit is not known, so we need to find it at job runtime. The Algorithm 1 shows the work-flow of ElasticYARN scheduler that perform the search and adapt the wave size. As soon as a client submits a job, the job watcher periodically notifies the scheduler and provides information about the job size and its progress (*line 1*). In lines 4 and 6 we calculate the number of containers to launch in the first wave of particular phase. For evaluation we set $p = 0.05$. If tasks in the first wave hit the limit, then only 5% of tasks will be affected. When the wave is complete we check if the limit is reached (*lines 8-9*) and calculate α. If the value of α indicates that we reach the limit, then we compute next wave size using equation 7. If the limit is not reached then increase the wave size (*line 15*)

7.6 EVALUATION

The goal of our evaluation is twofold. First, we want to estimate the quality of bandwidth cap estimation using our approach. Current version of YARN does not support container isolation for network I/O. Hence, network I/O is not a part of the scheduling algorithm. To enable the isolation we have to determine per container bandwidth cap that does not stretch the task running inside the container. Second, we want to apply ElasticYARN in two scenarios: inter-cloud deployment and cross-cloud deployment. The former assumes data storage is bottleneck, while in latter scenario network is bottleneck.

To evaluate ElasticYARN we implemented a testbed in our cluster consisting of 50 nodes connected via 1GB Ethernet link. Each of the nodes has 8 GB of RAM and 8 Intel Xeon E5405 CPUs. We configured NM of YARN to run up to 6 containers with the size of 1GB and 1 vcore. Other resources left to NM. From to 2 to 4 machines we dedicated for HDFS storage. For all experiments we set HDFS replication factor $R = 2$. One of the machines runs RM of YARN.

For the evaluation we took several types of MapReduce jobs, including Sort, Wordcount, Hive aggregation, Hive Join. The jobs have been used as main benchmarks in recent datacenter studies ([34, 71, 104, 156]). The size of the jobs presented in table 1.

```
1: Input: (Job = < M, R >, phase, progress)
2: if progress == 0 then
3:     if phase == 'map' then
4:         N = M * p
5:     else
6:         N = R * p
7:     end if
8: else
9:     if LimitFound == False then
10:        if IsWaveComplete() then
11:            α = getAlpha()
12:            if α < 1 then
13:                N = getLimit()
14:            else
15:                N = updateWave()
16:                LimitFound = True
17:            end if
18:        end if
19:    end if
20: end if
```

Algorithm 7: Job containers scaling

Job	Dataset Size	Maps	Reducers
WordCount	42 GB	240	180
Hive Join	25 GB	97	96
Hive Aggre	25 GB	97	96
Sort	42 GB	240	180

Table 1: MapReduce jobs used in the evaluation

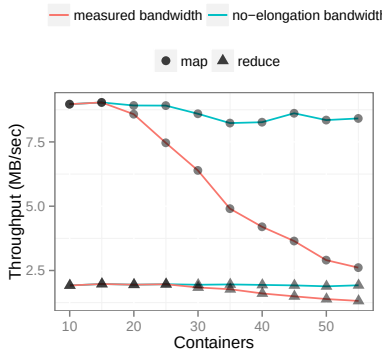


Figure 10: Sort

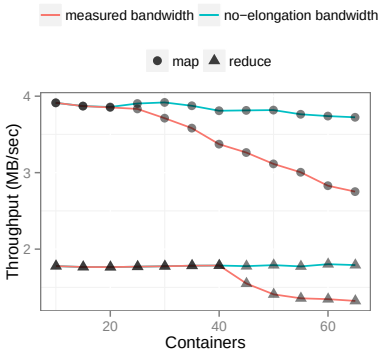


Figure 11: Hive aggregation

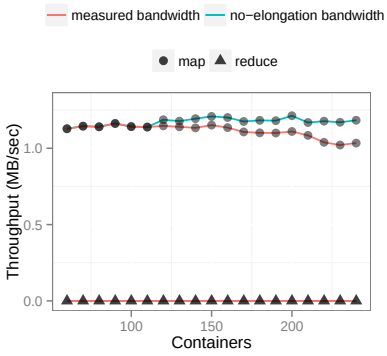


Figure 12: Wordcount

7.6.1 BANDWIDTH CAP ESTIMATION

We start our evaluation by varying number of containers running in a wave. The number of containers changes by 5. For this part of the evaluation we assigned only two nodes for the data storage. From each measurement we take the resource usage statistics of tasks that belong to the first wave of job execution and calculate no-elongation bandwidth cap. The cap is an upper limit that guarantees that task will not elongate due to saturation of non-scalable parts of EMR (data storage and network).

Graphs presented in figures 10, 11, 12, 16 show the measured and the no-elongation bandwidth of evaluated jobs. Starting from 20 containers map tasks (figure 10) of *Sort* job saturate the data storage. The measured bandwidth of tasks running in larger waves decreases. However, using our approach we can estimate the no-elongation bandwidth for each data point. We use equation 3 to estimate the value, which is about 8.5 MB/sec. The reduce tasks of Sort job reach the limit when the wave size goes above 30 containers. Similarly, we calculate the no-elongation bandwidth for the reduce task.

Figures 11, 16, 12 show the measured and the no-elongation bandwidth for other jobs. We can run up to 25 map tasks of *Hive aggregation* job, 120 map tasks of *Wordcount* job and 30 map tasks of *Hive join* job. Assume we use fixed size compute cluster with the capacity of 30 containers. Then the map phase of *Wordcount* needs 240/30 = 8 waves to finish, while it is possible to run 240/120 = 2 waves without elongation. Hence, we can finish the job earlier. The reduce phases of *wordcount* and *hive join* jobs do not hit the limit of the data storage. Therefore, in figures lines for the measured and the no-elongation bandwidth overlap.

The presented evaluation shows that we can calculate the no-elongation bandwidth for different MapReduce jobs using resource data from only one wave of the job execution.

7.6.2 RUNTIME CLUSTER RESIZING

In the second part of our evaluation we compare ElasticYARN with default YARN. We consider two scenarios. User can deploy EMR cluster either in a single cloud or run it in a cross-cloud fashion. We assume that in single cloud deployment the network is not a bottleneck, while the data storage is fixed at runtime. Hence, it can be saturated. We vary the data part capacity from 2 to 4 nodes. In the second scenario the data storage runs in one cloud and compute nodes on another cloud. For cross-cloud deployment we created two subnets in our 50-node cluster. The network is considered as bottleneck. We control the network bandwidth with *tc* command. *tc* is traffic control program for the Linux kernel.

First, we evaluate ElasticYARN in inter-cloud deployment. We compare ElasticYARN with default YARN equipped with 20, 40 and 60 compute nodes. ElasticYARN we run in two modes that update the wave size differently. In the first mode if the limit not found we increase the

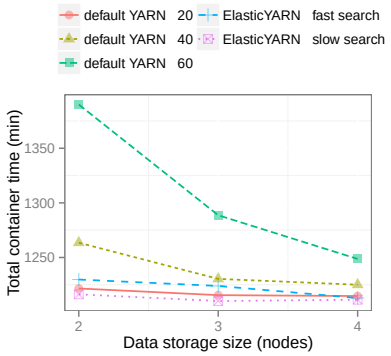


Figure 13: Total container time

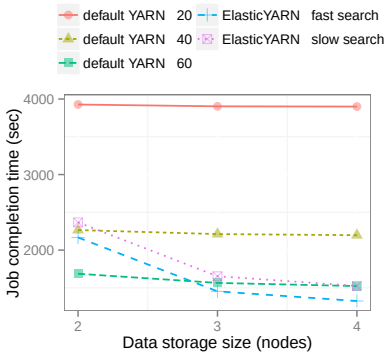


Figure 14: Total job time

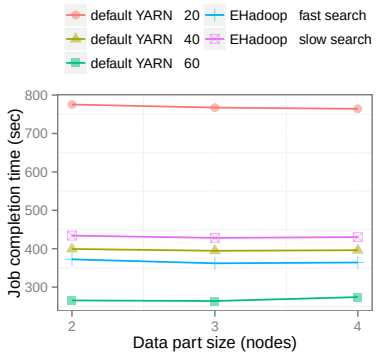


Figure 15: Wordcount completion time

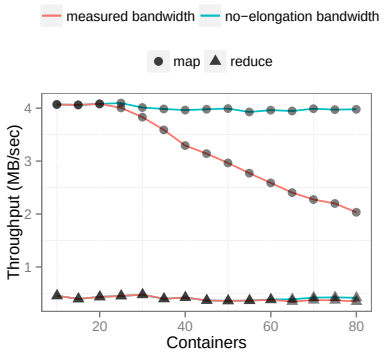
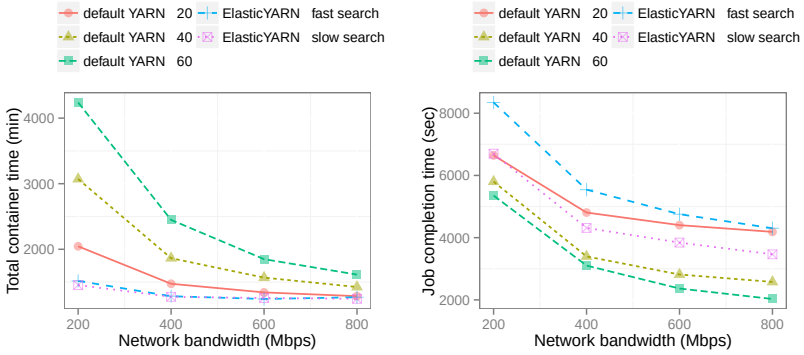


Figure 16: Hive join



(a) total container time

(b) total job time

Figure 17: Cross-cloud deployment

size of the wave by 5% of job size. For example, a job has 100 map tasks, then we start 5 tasks and in the next wave we run 10 tasks, if the limit is not found. However, if the limit is 40 tasks, then ElasticYARN will take up to 8 waves to find the limit, which is obviously increases a job completion time. Therefore we have another mode, where the wave size is increased by factor of two if the limit is not found. The first mode we call *slow search* and the second one *fast search*.

Figures 13, 14 present the total container time and the total job time. We summed up execution time of all benchmarks. ElasticYARN in both modes has the smallest total containers time in comparison to default YARN with 40 and 60 nodes. However, ElasticYARN in fast search mode running with the data storage that consists of 2 and 3 nodes has a bit higher than 20 nodes YARN the total container time. In this mode ElasticYARN doubles the wave size. Hence, in of the waves tasks saturate the data storage, before ElasticYARN resizes the wave according to the limit.

If we look at figure 14, then we see that ElasticYARN in both modes has almost 2 times shorter the total job time in comparison to YARN running with 20 nodes. With 3 and 4 nodes data storage ElasticYARN has same the total job completion time as YARN with 60 nodes. However, it incurs less costs (see figure 13). For some jobs such as *wordcount* the job completion time improvement in comparison to YARN with 20 nodes can be almost 2 times, which is show in figure 15.

In case of cross-cloud deployment the data between the storage and the compute nodes is transferred over the WAN. In figures 17a, 17b we present impact of evaluated deployments on the total container time and the job completion time. ElasticYARN has lowest the total container

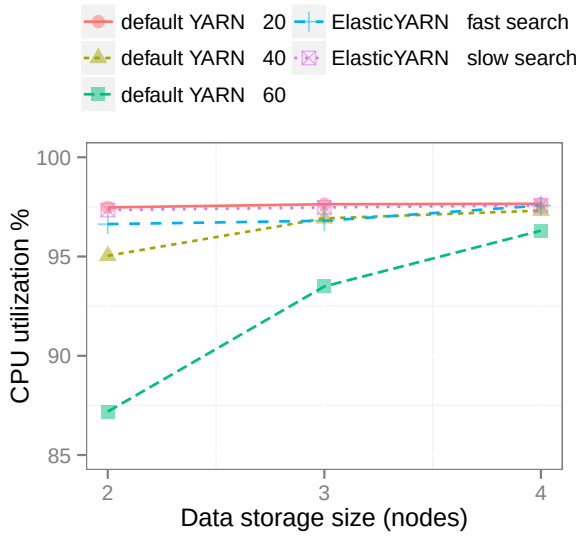


Figure 18: Inter-cloud deployment: container CPU utilization

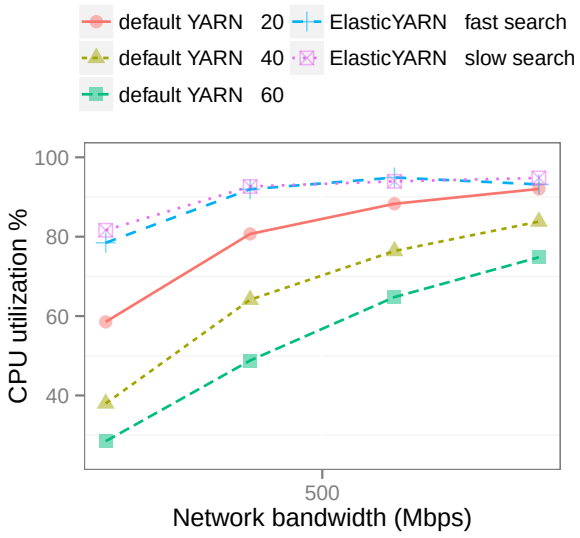


Figure 19: Cross-cloud deployment: container CPU utilization

time. However, the job completion time of the jobs running under control of ElasticYARN is longer than with 40, 60 nodes YARN. Since the network capacity is limited ElasticYARN shrinks size of waves. For example, with 200 Mbps network ElasticYARN runs only 1 map task of *Sort* job. ElasticYARN in slow search mode outperforms 20 nodes YARN.

ElasticYARN improves not only the total container time. It is also improves utilization of the containers. The utilization does not change, if we vary the data storage capacity. Figure 18 shows that in both modes ElasticYARN keeps the utilization above 95%. ElasticYARN adapts the number of containers running in parallel with respect to maximal performance provided by the storage. Without ElasticYARN the utilization can drop by 15% if we run YARN with 60 containers. In cross-cloud deployment the utilization of containers running with ElasticYARN is also about 95%. Only with 200 Mbps network it is about 80%. The utilization is lower, because the first wave already saturates the network.

7.7 RELATED WORK

Cluster sizing is not new problem. There have been a number of attempts to find optimal cluster size. Some works address scaling traditional data processing cluster, others only storage layer. In this work we focus on dynamic scaling support for ERM cluster.

The work by Jalaparti et al. [78] is mostly close to ours. The authors design Bazaar, a cloud framework offering a job-centric interface for data analytics applications. The job centric interface means that user specifies high-level goal, such as desired job completion time and Bazaar makes a decision about how many resources to allocate in order to achieve user's goal. The framework is focusing on two specific resources, compute instances and network bandwidth. Bazaar performs offline job profiling on dedicated node. Then the job profile is used for resource allocation decision. In our work we perform online job profiling on a life system. It allows us to reduce the job profiling time and avoid overhead associated with the setup of dedicated nodes and providing sample data, which is not always possible if the job appears for the first time.

Lim, Babu, and Chase [95] designed automated control for Elastic Storage. The scaling of the storage layer requires rebalancing persistent data across the nodes. The authors designed and implemented integral controller for HDFS. The controller targets 20% CPU utilization. The reference utilization allows to achieve average response time of 3 seconds. We see our work

as complimentary, since we provide dynamic scaling work compute layer of EMR.

Verma, Cherkasova, and Campbell [142] propose framework, called ARIA, to address the problem of resource allocation. Authors aim meet desired MapReduce job completion time. The system performs map-reduce slot allocation in heterogeneous environment. Similar to [78] ARIA requires offline job profiling to estimate required amount of slots for next execution of the job. Authors do not include the network bandwidth in scaling decisions. However, we observe, that the network can impact the task completion time.

Herodotou, Dong, and Babu [69] designed Elastisizer, a system to which users can express cluster sizing problems as queries in a declarative fashion. The system needs at least one run of the job to answer the user's query. In ElasticYARN we can make scaling decision from single wave of job execution.

Xie et al. [149] address the problem of shared datacenter network utilization. Authors analyzed traffic patterns of different Mapreduce jobs. And propose Proteus system that improves the network utilization. Similar to ElasticYARN, it detects the no-elongation bandwidth. However, it needs multiple job runs, before the bandwidth can be discovered.

7.8 CONCLUSION

Scaling the compute part of EMR cluster increases the traffic between compute and data nodes. If the number of compute nodes exceeds certain limit, then the MapReduce task completion stretches. As a result it incurs higher costs for the user. The elongation occurs, because the data nodes and/or the network cannot keep up with the increased demand. To solve the problem we presented ElasticYARN network I/O aware system for EMR cluster. ElasticYARN discovers the limit at the job runtime and calculates the number of tasks that can run in parallel without hitting the limit. We evaluate a ElasticYARN against set of MapReduce jobs. The evaluation shows that ElasticYARN provides minimal cost in case of varying capacity of the data storage and the network.

Current version of YARN does not support per container network I/O isolation. In the future we plan to integrate presented approach in YARN and include network I/O into RM scheduling decisions.

CONCLUSION

Capacity planning and dynamic resource scaling will be main topic of cloud computing in the near future. Both sides of cloud market interested in further improvement of existing resource allocation techniques. Provider revenue directly depends on the number of customers using cloud infrastructure. Hence, to accept more users IaaS providers either need to increase the number of datacenters or implement services that allocate resources with the minimal level of over-provisioning. Expanding datacenters is costly. It requires large upfront investment. There is also a need for nearby power station that has enough capacity to supply a datacenter. As a result, it limits cloud provider's options on locations of new datacenter.

Recent research states that utilization of modern datacenters is around 15 – 25%. It means that existing resource allocation techniques leave significant amount of resources under-utilized. There are two main reasons. The first reason is popular fixed size VM model. Capacity assigned to a VM does not change during runtime, while the application capacity demand can fluctuate. Moreover, VM templates usually defined on cloud provider's side. Hence, in order to avoid under-provisioning users are forced to allocate bigger VMs. It leads to resource wastage. Such inefficiency is covered by cloud users payment bills. The second reason is the lack of techniques for efficient model-free resource scaling. Cloud providers offer threshold based auto-scaling services. The services simplify virtual resource management process. But the task of scaling policy design is the user's responsibility. For a non-expert user it is a challenging task to implement an efficient scaling-policy. The user needs deep application knowledge and experience with underlying cloud infrastructure.

Economic interests of cloud users already resulted in changes on cloud market. There are public cloud providers that addressed the users expectations and shifted to flexible VM model. The users are free to specify a VM they need and can change it during runtime. Moreover, the providers start to move from hour billing cycles to second billing cycles. However, there is not much improvement regarding to scaling services. The focus of this thesis is to make one step forward to address cloud market changes and propose auto-scaling techniques for the users. In this work we design controllers that automatically perform scaling decisions to meet the application performance objectives and minimize the cost of virtual resources.

We start with an overview of the auto-scaling system design. Highlight the key phases of the auto-scaling process and each phase's role in the process. The quality of resource scaling decisions depends on the application performance model used in the auto-scaling system. The model describes quantitative relationships between the application virtual resource capacity, its performance and incoming workload. Design of the model requires expert knowledge and system identification experiments to catch the strength of correlation between aforementioned parameters. There is a set of techniques that is used in auto-scaling systems to describe the model. They are classified in five categories: threshold based, queuing theory, control theory, reinforcement learning and time series analysis. There is no *silver bullet* solution. Each of them has pros and cons. High resource allocation quality can be achieved with the combination of presented techniques.

In the following sections we summarize topics discussed in the chapters from 4 to 7. We also give our view on future research work that could be done with respect to approach presented in corresponding chapters.

8.1 VERTICAL SCALING FOR PRIORITIZED VMS PROVISIONING

In *Chapter 4* we presented time series based resource allocation controller. The controller exploits the idea of service differentiation. There are two types of applications running in modern data centers. The first type is latency-sensitive interactive applications. The examples are e-commerce web-sites, web-search engines, a web-based software office suite such Google Docs and etc. The second type of applications consists of resource intensive batch applications. Large body of these applications is presented by data analytic frameworks that use MapReduce

paradigm.

Interactive and batch applications react differently on resource under-provisioning. For interactive application it is important to have enough resources each moment of time, otherwise the latency goes up. In contrast, batch application can tolerate performance slowdown caused by resource shortage. To provide low latency we would need to over-provision interactive application, which leads to resource wastage. Therefore, we propose to collocate VMs running these applications on a host and resolve resource conflicts by scaling VMs vertically. The presented controller at first satisfies resource demand of the latency sensitive application and assigns residual resources to the batch application. Such service differentiation allows us to perform resource assignment with minimal over-provisioning and offers simple technique for the host utilization improvement.

8.1.1 FUTURE WORK

The controller provides performance guarantees only for interactive application. The assumption is that batch application user obtains compute resources for a lower price and aware of possible performance slowdown. One potential direction for future research is providing job completion time guarantees for low priority batch applications. Usually batch applications run across a number of machines. Hence, we can select hosts which cumulative amount of expected available resources is enough to finish a job within user-specified deadline.

8.2 REINFORCEMENT LEARNING BASED TECHNIQUES

8.2.1 AUTONOMIC VIRTUAL MACHINE SCALING

In *Chapter 5* and *Chapter 6* we exploit reinforcement learning approach to describe application performance model and time series to anticipate incoming workload. Reinforcement learning offers knowledge-free learning algorithms. It eliminates the need for offline application model design. The scaling policy evolves online with the help of trial-and-error approach. However, the learning process can take significant amount of time. Therefore early stage scaling decisions of RL based auto-scaling systems are non-optimal.

To improve the learning process time in *Chapter 5* we propose a speedup technique. The learning agent after each scaling action updates only one state-action transition. However, we observe that for resource allocation problem there is more data learn. Usually during initialization phase resource allocation is performed with some level of over-provisioning. It enables a number of alternative states that can be visited by the learning agent. Transitions that lead to the alternative states can be also updated. Hence, more than one transition can be updated after each resource allocation action. Our evaluation shows that presented technique significantly improves learning time without affecting the quality of scaling decisions.

8.2.2 AUTONOMIC MULTI-TIER APPLICATION SCALING

The second well-known problem of reinforcement learning approaches is the curse of dimensionality. The state-space dramatically grows with increased number of parameters that describe the model of the application. It is common to reduce state-space and actions-space to address the issue. However, it leads to coarse-granular resource allocation. One of the goals of resource allocation is providing performance of the application with respect to user-specified objective. In *Chapter 6* we analyzed impact of individual VM resources on multi-tier web-application performance. We found that only CPU smoothly regulates the application response time. We created two separate models of each tier of the application. The approach reduces the state-space complexity. To orchestrate the models of the web application tiers

we added workload description parameter. The evaluation presented in *Chapter 6* shows that applied approach allows to efficiently allocate resources to the application and satisfy user's SLO.

8.2.3 FUTURE WORK

The presented RL based controllers use vertical scaling, which is limited by a host capacity. For larger workloads we would need to expand beyond single host. In the future we want to combine vertical and horizontal scaling to serve the workloads. Such combination creates a large number of options for possible resource assignment. For example, we can allocate 4 VMs with 1 GB RAM or 2 VMs with 2 GB RAM, or even 1 VM with 4 GB of RAM. This is only for one of resources. We can also control memory, disk I/O and network I/O. To address the problem, we could think about two level controller. The first level estimates the impact of vertical scaling on the application performance. The second level makes decision about optimal combination of vertical and horizontal scaling.

8.3 ELASTIC MAPREDUCE CLUSTER SCALING

In contrast to interactive applications the resource allocation of batch application denoted as scheduling. Batch application usually runs as a set of tasks on a number of machines that compose a cluster. One of the problems is to determine the size of the cluster to meet job completion deadline. For these applications it is intuitive to map resource allocation to performance. For example, to reduce job completion time by factor of two one would increase the cluster size also by factor of two. It is true for resources such as CPU and memory, because existing cluster management frameworks run a task of batch application inside a container that isolates only these resources. However, in case of network I/O resources it is more complicated.

We show that increasing number of compute nodes in elastic MapReduce cluster increases a total container time as well. The total container time is the time that accounts as resource usage time in the cloud environment. In the cloud users pay for the time a resource being used. Hence, we need to determine the cluster size that in given conditions provides the minimal total container time.

To find optimal cluster size in case of network I/O bottlenecks one needs to run a job multiple times. In *Chapter 7* we analyzed MapReduce application execution model and discover resource usage patterns that we can use to quantify the impact of network I/O related bottlenecks on different MapReduce jobs. We implemented controller called ElasticYARN that finds optimal cluster size at job runtime. Our evaluation shows that the controller provides the minimal total container time in inter-cloud and cross-cloud scenarios.

8.3.1 FUTURE WORK

YARN framework uses containers to isolate CPU and memory resources assigned to a task. However, the container technology also supports I/O resources isolation. In *Chapter 7* we show that ElasticYARN determines MapReduce job's task bandwidth requirements at runtime. In future we want apply our approach in YARN scheduler to make it network I/O aware.

BIBLIOGRAPHY

[1] Giuseppe Aceto et al. "Survey Cloud Monitoring: A Survey". In: *Comput. Netw.* 57.9 (June 2013), pp. 2093–2115. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2013.04.001. URL: http://dx.doi.org/10.1016/j.comnet.2013.04.001 (cit. on p. 31).

[2] O. Agmon Ben-Yehuda et al. "Deconstructing Amazon EC2 Spot Instance Pricing". In: *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. 2011, pp. 304–311. DOI: 10.1109/CloudCom.2011.48 (cit. on p. 26).

[3] Orna Agmon Ben-Yehuda et al. "Ginseng: Market-driven Memory Allocation". In: *SIGPLAN Not.* 49.7 (Mar. 2014), pp. 41–52. ISSN: 0362-1340. DOI: 10.1145/2674025.2576197. URL: http://doi.acm.org/10.1145/2674025.2576197 (cit. on p. 26).

[4] Orna Agmon Ben-Yehuda et al. "The Rise of RaaS: The Resource-as-a-service Cloud". In: *Commun. ACM* 57.7 (July 2014), pp. 76–84. ISSN: 0001-0782. DOI: 10.1145/2627422. URL: http://doi.acm.org/10.1145/2627422 (cit. on pp. 20, 25, 63, 93).

[5] Faraz Ahmad et al. "ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 1–13. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ahmad (cit. on p. 21).

[6] Faraz Ahmad et al. "ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 1–12. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643636 (cit. on p. 25).

[7] A. Ali-Eldin, J. Tordsson, and E. Elmroth. "An adaptive hybrid elasticity controller for cloud infrastructures". In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. 2012, pp. 204–212. DOI: 10.1109/NOMS.2012.6211900 (cit. on p. 42).

[8] Ahmed Ali-Eldin et al. "Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control". In: *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*. ScienceCloud '12. Delft, The Netherlands: ACM, 2012, pp. 31–40. ISBN: 978-1-4503-1340-7. DOI: 10.1145/2287036.2287044. URL: http://doi.acm.org/10.1145/2287036.2287044 (cit. on pp. 20, 42).

[9] *Amazon auto scaling service*. URL: http://aws.amazon.com/autoscaling (visited on 07/18/2013) (cit. on pp. 49, 51).

[10] *Amazon EC2*. URL: https://cloud.google.com/compute (visited on 05/10/2014) (cit. on p. 17).

[11] *Amazon EC2 compute unit*. URL: https://huanliu.wordpress.com/2010/06/14/amazons-physical-hardware-and-ec2-compute-unit (visited on 07/30/2015) (cit. on pp. 22, 86).

[12] *Amazon Elastic MapReduce*. URL: http://aws.amazon.com/elasticmapreduce (visited on 08/10/2013) (cit. on p. 93).

[13] Ganesh Ananthanarayanan et al. "Disk-locality in Datacenter Computing Considered Irrelevant". In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. Napa, California: USENIX Association, 2011, pp. 12–12. URL: http://dl.acm.org/citation.cfm?id=1991596.1991613 (cit. on p. 25).

[14] Ganesh Ananthanarayanan et al. "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters". In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 287–300. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966472. URL: http://doi.acm.org/10.1145/1966445.1966472 (cit. on p. 25).

[15] *Apache Hadoop-based service in the cloud from Microsoft Azure*. URL: https://azure.microsoft.com/en-us/services/hdinsight (visited on 05/10/2015) (cit. on p. 95).

[16] *Applications and Organizations using Hadoop*. URL: http://wiki.apache.org/hadoop/PoweredBy (visited on 06/20/2015) (cit. on p. 24).

[17] Michael Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672 (cit. on pp. 26, 40).

[18] Ilia Baldine et al. "The missing link: Putting the network in networked cloud computing". In: *in ICVCI09: International Conference on the Virtual Computing Initiative*. 2009 (cit. on p. 21).

[19] Paul Barham et al. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462. URL: http://doi.acm.org/10.1145/945445.945462 (cit. on p. 21).

[20] Enda Barrett, Enda Howley, and Jim Duggan. "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud". In: *Concurrency and Computation: Practice and Experience* 25.12 (2013), pp. 1656–1674 (cit. on pp. 33, 34, 37, 38, 64, 65, 67).

[21] Dominique Bellenger et al. "Scaling in cloud environments". In: *Proceedings of the 15th WSEAS international conference on Computers*. Corfu Island, Greece: World Scientific, Engineering Academy, and Society (WSEAS), 2011, pp. 145–150. ISBN: 978-1-61804-019-0. URL: http://dl.acm.org/citation.cfm?id=2028299.2028329 (cit. on p. 50).

[22] Peter Bodik et al. "Automatic Exploration of Datacenter Performance Regimes". In: *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*. ACDC '09. Barcelona, Spain: ACM, 2009, pp. 1–6. ISBN: 978-1-60558-585-7. DOI: 10.1145/1555271.1555273. URL: http://doi.acm.org/10.1145/1555271.1555273 (cit. on p. 33).

[23] Roy Bryant et al. "Kaleidoscope: Cloud Micro-elasticity via VM State Coloring". In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 273–286. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966471. URL: http://doi.acm.org/10.1145/1966445.1966471 (cit. on p. 20).

[24] Xiangping Bu, Jia Rao, and Cheng zhong Xu. "Coordinated Self-Configuration of Virtual Machines and Appliances Using a Model-Free Learning Approach". In: *Parallel and Distributed Systems, IEEE Transactions on* 24.4 (2013), pp. 681–690. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.174 (cit. on p. 84).

[25] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. "CoTuner: A Framework for Coordinated Auto-configuration of Virtualized Resources and Appliances". In: *Proceedings of the 7th International Conference on Autonomic Computing*. ICAC '10. Washington, DC, USA: ACM, 2010, pp. 75–76. ISBN: 978-1-4503-0074-2. DOI: 10.1145/1809049.1809062. URL: http://doi.acm.org/10.1145/1809049.1809062 (cit. on p. 38).

[26] Emiliano Casalicchio and Luca Silvestri. "Mechanisms for {SLA} provisioning in cloud-based service providers". In: *Computer Networks* 57.3 (2013), pp. 795 –810. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2012.10.020. URL: http://www.sciencedirect.com/science/article/pii/S1389128612003763 (cit. on pp. 20, 31, 34, 35, 93).

[27] A. Celesti et al. "How to Enhance Cloud Architectures to Enable Cross-Federation". In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. 2010, pp. 337–345. DOI: 10.1109/CLOUD.2010.46 (cit. on p. 20).

[28] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. "Dynamic Resource Allocation for Shared Data Centers Using Online Measurements". In: *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '03. San Diego, CA, USA: ACM, 2003, pp. 300–301. ISBN: 1-58113-664-1. DOI: 10.1145/781027.781067. URL: http://doi.acm.org/10.1145/781027.781067 (cit. on p. 43).

[29] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. "Comparison of the Three CPU Schedulers in Xen". In: *SIGMETRICS Perform. Eval. Rev.* 35.2 (Sept. 2007), pp. 42–51. ISSN: 0163-5999. DOI: 10.1145/1330555.1330556. URL: http://doi.acm.org/10.1145/1330555.1330556 (cit. on p. 21).

[30] Brian Cho et al. "Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in Mapreduce Clusters". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 6:1–6:17. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523624. URL: http://doi.acm.org/10.1145/2523616.2523624 (cit. on p. 24).

[31] Christopher Clark et al. "Live Migration of Virtual Machines". In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286. URL: http://dl.acm.org/citation.cfm?id=1251203.1251223 (cit. on p. 20).

[32] R. Benjamin Clay, Zhiming Shen, and Xiaosong Ma. "Accelerating Batch Analytics with Residual Resources from Interactive Clouds". In: *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. MASCOTS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 414–423. ISBN: 978-0-7695-5102-9. DOI: 10.1109/MASCOTS.2013.63. URL: http://dx.doi.org/10.1109/MASCOTS.2013.63 (cit. on pp. 24, 25).

[33] *CloudSigma IaaS provider*. URL: http://www.cloudsigma.com (visited on 01/10/2013) (cit. on pp. 26, 63, 64, 77).

[34] Tyson Condie et al. "Online Aggregation and Continuous Query Support in MapReduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 1115–1118. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807295. URL: http://doi.acm.org/10.1145/1807167.1807295 (cit. on p. 102).

[35] James C. Corbett et al. "Spanner: Google&Rsquo;s Globally Distributed Database". In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013), 8:1–8:22. ISSN: 0734-2071. DOI: 10.1145/2491245. URL: http://doi.acm.org/10.1145/2491245 (cit. on p. 95).

[36] Paolo Costa et al. "Camdoop: Exploiting In-network Aggregation for Big Data Applications". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=2228298.2228302 (cit. on p. 25).

[37] *Credit-Based CPU Scheduler*. URL: http://wiki.xen.org (visited on 03/17/2012) (cit. on pp. 51, 65, 66, 79).

[38] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. "Elastic virtual machine for fine-grained cloud resource provisioning". In: *Global Trends in Computing and Communication Systems*. Springer, 2012, pp. 11–25 (cit. on p. 42).

[39] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. "Elastic VM for Cloud Resources Provisioning Optimization". In: *Advances in Computing and Communications - First International Conference, ACC 2011, Kochi, India, July 22-24, 2011. Proceedings, Part I*. Ed. by Ajith Abraham et al. Vol. 190. Communications in Computer and Information Science. Springer, 2011, pp. 431–445. ISBN: 978-3-642-22708-0. DOI: 10.1007/978-3-642-22709-7_43. URL: http://dx.doi.org/10.1007/978-3-642-22709-7_43 (cit. on pp. 22, 23).

[40] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264 (cit. on pp. 24, 56, 93, 94).

[41] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541941. URL: http://doi.acm.org/10.1145/2541940.2541941 (cit. on pp. 17, 18, 23, 25, 33, 49, 94).

[42] Peter A. Dinda and David R. O'Hallaron. "Host load prediction using linear models". In: *Cluster Computing* 3 (4 2000), pp. 265–280. ISSN: 1386-7857. DOI: 10.1023/A:1019048724544. URL: http://dl.acm.org/citation.cfm?id=592893.592958 (cit. on p. 52).

[43] Ronald P. Doyle et al. "Model-based Resource Provisioning in a Web Service Utility". In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS'03. Seattle, WA: USENIX Association, 2003, pp. 5–5. URL: http://dl.acm.org/citation.cfm?id=1251460.1251465 (cit. on p. 32).

[44] Abhishek Dubey et al. "Performance Modeling of Distributed Multi-tier Enterprise Systems". In: *SIGMETRICS Perform. Eval. Rev.* 37.2 (Oct. 2009), pp. 9–11. ISSN: 0163-5999. DOI: 10.1145/1639562.1639566. URL: http://doi.acm.org/10.1145/1639562.1639566 (cit. on p. 58).

[45] X. Dutreilh et al. "From Data Center Resource Allocation to Control Theory and Back". In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. 2010, pp. 410–417. DOI: 10.1109/CLOUD.2010.55 (cit. on pp. 35, 40).

[46] Xavier Dutreilh et al. "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow". In: *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*. 2011, pp. 67–74 (cit. on pp. 37, 64, 67).

[47] Sourav Dutta et al. "SmartScale: Automatic Application Scaling in Enterprise Clouds". In: *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*. CLOUD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 221–228. ISBN: 978-0-7695-4755-8. DOI: 10.1109/CLOUD.2012.12. URL: http://dx.doi.org/10.1109/CLOUD.2012.12 (cit. on pp. 20, 32, 50, 65, 68, 73, 87).

[48] Andy Edmonds et al. "FluidCloud: An Open Framework for Relocation of Cloud Services". In: *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. San Jose, CA: USENIX, 2013. URL: https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Edmonds (cit. on pp. 93, 95).

[49] Vincent C. Emeakaroha et al. "Towards Autonomic Detection of SLA Violations in Cloud Infrastructures". In: *Future Gener. Comput. Syst.* 28.7 (July 2012), pp. 1017–1029. ISSN: 0167-739X. DOI: 10.1016/j.future.2011.08.018. URL: http://dx.doi.org/10.1016/j.future.2011.08.018 (cit. on p. 31).

[50] Wei Fang et al. "RPPS: A Novel Resource Prediction and Provisioning Scheme in Cloud Data Center". In: *Services Computing (SCC), 2012 IEEE Ninth International Conference on*. 2012, pp. 609–616. DOI: 10.1109/SCC.2012.47 (cit. on p. 44).

[51] Andrew D. Ferguson et al. "Jockey: Guaranteed Job Latency in Data Parallel Clusters". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 99–112. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168847. URL: http://doi.acm.org/10.1145/2168836.2168847 (cit. on p. 40).

[52] Alessio Gambi et al. "On Estimating Actuation Delays in Elastic Computing Systems". In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 33–42. ISBN: 978-1-4673-4401-2. URL: http://dl.acm.org/citation.cfm?id=2487336.2487345 (cit. on p. 32).

[53] Anshul Gandhi et al. "Adaptive, Model-driven Autoscaling for Cloud Applications". In: *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 57–64. ISBN: 978-1-931971-11-9. URL: https://www.usenix.org/conference/icac14/technical-sessions/presentation/gandhi (cit. on pp. 41, 42, 78).

[54] Thomas Gleixner, Paul E. McKenney, and Vincent Guittot. "Cleaning Up Linux's CPU Hotplug for Real Time and Energy Management". In: *SIGBED Rev.* 9.4 (Nov. 2012), pp. 49–52. ISSN: 1551-3688. DOI: 10.1145/2452537.2452547. URL: http://doi.acm.org/10.1145/2452537.2452547 (cit. on p. 21).

[55] Daniel Gmach, Jerry Rolia, and Ludmila Cherkasova. "Selling T-shirts and Time Shares in the Cloud". In: *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 539–546. ISBN: 978-0-7695-4691-9. DOI: 10.1109/CCGrid.2012.68. URL: http://dx.doi.org/10.1109/CCGrid.2012.68 (cit. on p. 20).

[56] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. "PRESS: PRedictive Elastic ReSource Scaling for cloud systems". In: *Network and Service Management (CNSM), 2010 International Conference on*. 2010, pp. 9–16. DOI: 10.1109/CNSM.2010.5691343 (cit. on pp. 34, 43, 44, 51, 87).

[57] *Google App Engine*. URL: http://code.google.com/appengine (visited on 12/23/2014) (cit. on p. 17).

[58] *Google cloud platform*. URL: https://cloud.google.com/compute (visited on 05/14/2014) (cit. on p. 17).

[59] *GridSpot IaaS provider*. URL: http://www.gridspot.com (visited on 02/10/2013) (cit. on pp. 26, 63).

[60] Yanfei Guo, P. Lama, and Xiaobo Zhou. "Automated and Agile Server Parameter Tuning with Learning and Control". In: *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. 2012, pp. 656–667. DOI: 10.1109/IPDPS.2012.66 (cit. on p. 38).

[61] Yanfei Guo and Xiaobo Zhou. "Coordinated VM Resizing and Server Tuning: Throughput, Power Efficiency and Scalability". In: *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*. 2012, pp. 289–297. DOI: 10.1109/MASCOTS.2012.41 (cit. on p. 38).

[62] Rui Han et al. "Enabling Cost-aware and Adaptive Elasticity of Multi-tier Cloud Applications". In: *Future Gener. Comput. Syst.* 32 (Mar. 2014), pp. 82–98. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.05.018. URL: http://dx.doi.org/10.1016/j.future.2012.05.018 (cit. on p. 42).

[63] Rui Han et al. "Lightweight Resource Scaling for Cloud Applications". In: *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 644–651. ISBN: 978-0-7695-4691-9. DOI: 10.1109/CCGrid.2012.52. URL: http://dx.doi.org/10.1109/CCGrid.2012.52 (cit. on pp. 24, 34, 35).

[64] M.Z. Hasan et al. "Integrated and autonomic cloud resource scaling". In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. 2012, pp. 1327–1334. DOI: 10.1109/NOMS.2012.6212070 (cit. on pp. 34–36).

[65] J.L. Hellerstein, Yixin Diao, and S. Parekh. "A first-principles approach to constructing transfer functions for admission control in computing systems". In: *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*. Vol. 3. 2002, 2906–2912 vol.3. DOI: 10.1109/CDC.2002.1184291 (cit. on p. 39).

[66] J.L. Hellerstein, S. Singhal, and Qian Wang. "Research challenges in control engineering of computing systems". In: *Network and Service Management, IEEE Transactions on* 6.4 (2009), pp. 206–211. ISSN: 1932-4537. DOI: 10.1109/TNSM.2009.04.090401 (cit. on p. 39).

[67] J. Heo et al. "Memory overbooking and dynamic control of Xen virtual machines in consolidated environments". In: *Integrated Network Management, 2009. IM '09. IFIP/IEEE International Symposium on*. 2009, pp. 630–637. DOI: 10.1109/INM.2009.5188871 (cit. on p. 32).

[68] Jin Heo et al. "Memory Overbooking and Dynamic Control of Xen Virtual Machines in Consolidated Environments". In: *Proceedings of the 11th IFIP/IEEE International Conference on Symposium on Integrated Network Management*. IM'09. New York, NY, USA: IEEE Press, 2009, pp. 630–637. ISBN: 978-1-4244-3486-2. URL: http://dl.acm.org/citation.cfm?id=1688933.1689025 (cit. on pp. 22, 40, 72, 87).

[69] Herodotos Herodotou, Fei Dong, and Shivnath Babu. "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics". In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 2011, 18:1–18:14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038934. URL: http://doi.acm.org/10.1145/2038916.2038934 (cit. on pp. 94, 107).

[70] Zach Hill et al. "Early observations on the performance of Windows Azure". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. Chicago, Illinois: ACM, 2010, pp. 367–376. ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851532. URL: http://doi.acm.org/10.1145/1851476.1851532 (cit. on p. 50).

[71] Benjamin Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488 (cit. on pp. 94, 102).

[72] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. Morgan and Claypool Publishers, 2009. ISBN: 159829556X, 9781598295566 (cit. on p. 25).

[73] Anca Iordache et al. "Resilin: Elastic MapReduce over Multiple Clouds". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* 0 (2013), pp. 261–268. DOI: http://doi.ieeecomputersociety.org/10.1109/CCGrid.2013.48 (cit. on pp. 93, 95).

[74] Waheed Iqbal, Matthew Dailey, and David Carrera. "SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud". In: *Proceedings of the 1st International Conference on Cloud Computing*. CloudCom '09. Beijing, China: Springer-Verlag, 2009, pp. 243–253. ISBN: 978-3-642-10664-4. DOI: 10.1007/978-3-642-10665-1_22. URL: http://dx.doi.org/10.1007/978-3-642-10665-1_22 (cit. on p. 34).

[75] C. Isci et al. "Improving server utilization using fast virtual machine migration". In: *IBM Journal of Research and Development* 55.6 (2011), 4:1–4:12. ISSN: 0018-8646. DOI: 10.1147/JRD.2011.2167775 (cit. on p. 20).

[76] Sadeka Islam et al. "Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud". In: *Future Gener. Comput. Syst.* 28.1 (Jan. 2012), pp. 155–162. ISSN: 0167-739X. DOI: 10.1016/j.future.2011.05.027. URL: http://dx.doi.org/10.1016/j.future.2011.05.027 (cit. on p. 44).

[77] Bart Jacob et al. *A practical guide to the IBM autonomic computing toolkit*. 2004 (cit. on p. 31).

[78] Virajith Jalaparti et al. "Bridging the Tenant-provider Gap in Cloud Services". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 10:1–10:14. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391239. URL: http://doi.acm.org/10.1145/2391229.2391239 (cit. on pp. 106, 107).

[79] Jing Jiang et al. "Optimal Cloud Resource Auto-Scaling for Web Applications". In: *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. 2013, pp. 58–65. DOI: 10.1109/CCGrid.2013.73 (cit. on p. 43).

[80] *Joyent Cloud*. URL: http://www.joyentcloud.com (visited on 03/17/2013) (cit. on p. 51).

[81] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. "Reinforcement Learning: A Survey". In: *J. Artif. Int. Res.* 4.1 (May 1996), pp. 237–285. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1622737.1622748 (cit. on p. 36).

[82] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. "Adaptive Resource Provisioning for Virtualized Servers Using Kalman Filters". In: *ACM Trans. Auton. Adapt. Syst.* 9.2 (July 2014), 10:1–10:35. ISSN: 1556-4665. DOI: 10.1145/2626290. URL: http://doi.acm.org/10.1145/2626290 (cit. on pp. 24, 43, 78).

[83] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. "Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters". In: *Proceedings of the 6th International Conference on Autonomic Computing*. ICAC '09. Barcelona, Spain: ACM, 2009, pp. 117–126. ISBN: 978-1-60558-564-2. DOI: 10.1145/1555228.1555261. URL: http://doi.acm.org/10.1145/1555228.1555261 (cit. on pp. 40, 58, 78, 87).

[84] David G Kendall. "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain". In: *The Annals of Mathematical Statistics* (1953), pp. 338–354 (cit. on p. 41).

[85] Pawel Koperek and Wlodzimierz Funika. "Dynamic Business Metrics-driven Resource Provisioning in Cloud Environments". In: *Parallel Processing and Applied Mathematics - 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part II*. Ed. by Roman Wyrzykowski et al. Vol. 7204. Lecture Notes in Computer Science. Springer, 2011, pp. 171–180. ISBN: 978-3-642-31499-5. DOI: 10.1007/978-3-642-31500-8_18. URL: http://dx.doi.org/10.1007/978-3-642-31500-8_18 (cit. on pp. 31, 34, 35).

[86] R. Matthew Kretchmar. "Parallel Reinforcement Learning". In: *In The 6th World Conference on Systemics, Cybernetics, and Informatics*. 2002 (cit. on p. 65).

[87] Gautam Kumar et al. "A Case for Performance-centric Network Allocation". In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. HotCloud'12. Boston, MA: USENIX Association, 2012, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=2342763.2342772 (cit. on pp. 21, 93, 95, 98).

[88] Sajib Kundu et al. "Modeling Virtualized Applications Using Machine Learning Techniques". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. London, England, UK: ACM, 2012, pp. 3–14. ISBN: 978-1-4503-1176-2. DOI: 10.1145/2151024.2151028. URL: http://doi.acm.org/10.1145/2151024.2151028 (cit. on pp. 44, 87, 88).

[89] J Kupferman et al. "Scaling into the cloud,(2009)". In: *URL http://cs. ucsb. edu/~ jkupferman/docs/ScalingIntoTheClouds. pdf (available online accessed 29.01. 12)* () (cit. on pp. 35, 93).

[90] Horacio Andrés Lagar-Cavilla et al. "SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 1–12. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519067. URL: http://doi.acm.org/10.1145/1519065.1519067 (cit. on p. 20).

[91] P. Lama and Xiaobo Zhou. "Autonomic Provisioning with Self-Adaptive Neural Fuzzy Control for End-to-end Delay Guarantee". In: *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. 2010, pp. 151–160. DOI: 10.1109/MASCOTS.2010.24 (cit. on p. 40).

[92] Palden Lama and Xiaobo Zhou. "AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud". In: *Proceedings of the 9th International Conference on Autonomic Computing*. ICAC '12. San Jose, California, USA: ACM, 2012, pp. 63–72. ISBN: 978-1-4503-1520-3. DOI: 10.1145/2371536.2371547. URL: http://doi.acm.org/10.1145/2371536.2371547 (cit. on p. 44).

[93] Jingshan Li and Semyon M Meerkov. *Production systems engineering*. Springer Science & Business Media, 2008 (cit. on p. 41).

[94] Min Li et al. "MRONLINE: MapReduce Online Performance Tuning". In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: ACM, 2014, pp. 165–176. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600229. URL: http://doi.acm.org/10.1145/2600212.2600229 (cit. on pp. 22, 24).

[95] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. "Automated Control for Elastic Storage". In: *Proceedings of the 7th International Conference on Autonomic Computing*. ICAC '10. Washington, DC, USA: ACM, 2010, pp. 1–10. ISBN: 978-1-4503-0074-2. DOI: 10.1145/1809049.1809051. URL: http://doi.acm.org/10.1145/1809049.1809051 (cit. on pp. 35, 40, 94, 106).

[96] Harold C. Lim et al. "Automated Control in Cloud Computing: Challenges and Opportunities". In: *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*. ACDC '09. Barcelona, Spain: ACM, 2009, pp. 13–18. ISBN: 978-1-60558-585-7. DOI: 10.1145/1555271.1555275. URL: http://doi.acm.org/10.1145/1555271.1555275 (cit. on p. 40).

[97] Bin Lin and Peter A. Dinda. "VSched: Mixing Batch And Interactive Virtual Machines Using Periodic Real-time Scheduling". In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 8–. ISBN: 1-59593-061-2. DOI: 10.1109/SC.2005.80. URL: http://dx.doi.org/10.1109/SC.2005.80 (cit. on pp. 58, 59).

[98] Jie Liu et al. "The Data Furnace: Heating Up with Cloud Computing". In: *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'11. Portland, OR: USENIX Association, 2011, pp. 15–15. URL: http://dl.acm.org/citation.cfm?id=2170444.2170459 (cit. on p. 95).

[99] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. "Comparison of Auto-scaling Techniques for Cloud Environments". In: *Actas de las XXIV Jornadas de Paralelismo*. Ed. by Guillermo Botella y Alberto A. Del Barrio. Servicio de Publicaciones. Universidad Complutense de Madrid, 2013. ISBN: 978-84-695-8330-2 (cit. on p. 36).

[100] Marissa Mayer. "In search of a better, faster, stronger web". In: *Proc. Velocity* (2009) (cit. on pp. 23, 33).

[101] *Microsoft Azure Services*. URL: http://azure.microsoft.com (visited on 02/14/2015) (cit. on p. 17).

[102] Murtaza Motiwala et al. "Path Splicing". In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: ACM, 2008, pp. 27–38. ISBN: 978-1-60558-175-0. DOI: 10.1145/1402958.1402963. URL: http://doi.acm.org/10.1145/1402958.1402963 (cit. on p. 21).

[103] *Moving ahead with Hadoop YARN*. URL: http://www.ibm.com/developerworks/library/bd-hadoopyarn/ (visited on 01/30/2015) (cit. on p. 96).

[104] Derek G. Murray et al. "CIEL: A Universal Execution Engine for Distributed Data-flow Computing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 113–126. URL: http://dl.acm.org/citation.cfm?id=1972457.1972470 (cit. on p. 102).

[105] Mohamed N. Bennani and Daniel A. Menasce. "Resource Allocation for Autonomic Data Centers Using Analytic Performance Models". In: *Proceedings of the Second International Conference on Automatic Computing*. ICAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 229–240. ISBN: 0-7965-2276-9. DOI: 10.1109/ICAC.2005.50. URL: http://dx.doi.org/10.1109/ICAC.2005.50 (cit. on p. 43).

[106] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. "Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 237–250. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755938. URL: http://doi.acm.org/10.1145/1755913.1755938 (cit. on pp. 26, 40).

[107] Hiep Nguyen et al. "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service". In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 69–82. ISBN: 978-1-931971-02-7. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/nguyen (cit. on pp. 20, 44).

[108] Pradeep Padala et al. "Adaptive Control of Virtualized Resources in Utility Computing Environments". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 289–302. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273026. URL: http://doi.acm.org/10.1145/1272996.1273026 (cit. on pp. 31, 32, 39, 40, 58, 63, 78, 87).

[109] Pradeep Padala et al. "Automated Control of Multiple Virtualized Resources". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 13–26. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519068. URL: http://doi.acm.org/10.1145/1519065.1519068 (cit. on pp. 26, 40, 72, 87).

[110] Sankaralingam Panneerselvam and Michael M. Swift. "Chameleon: Operating System Support for Dynamic Processors". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 99–110. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150988. URL: http://doi.acm.org/10.1145/2150976.2150988 (cit. on pp. 21, 50).

[111] Sang-Min Park and Marty Humphrey. "Self-Tuning Virtual Machines for Predictable eScience". In: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. CCGRID '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 356–363. ISBN: 978-0-7695-3622-4. DOI: 10.1109/CCGRID.2009.84. URL: http://dx.doi.org/10.1109/CCGRID.2009.84 (cit. on p. 43).

[112] Tharindu Patikirikorala and Alan Colman. "Feedback controllers in the cloud". In: *Proceedings of APSEC*. 2010 (cit. on p. 39).

[113] *Profitbricks IaaS provider*. URL: http://www.profitbricks.com (visited on 03/22/2013) (cit. on pp. 26, 63).

[114] Do Le Quoc, Lenar Yazdanov, and Christof Fetzer. "DoLen: User-Side Multi-cloud Application Monitoring". In: *Proceedings of the 2014 International Conference on Future Internet of Things and Cloud*. FICLOUD '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 76–81. ISBN: 978-1-4799-4357-9. DOI: 10.1109/FiCloud.2014.22. URL: http://dx.doi.org/10.1109/FiCloud.2014.22 (cit. on p. 101).

[115] Barath Raghavan et al. "Cloud Control with Distributed Rate Limiting". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Kyoto, Japan: ACM, 2007, pp. 337–348. ISBN: 978-1-59593-713-1. DOI: 10.1145/1282380.1282419. URL: http://doi.acm.org/10.1145/1282380.1282419 (cit. on p. 21).

[116] Jia Rao et al. "A Distributed Self-Learning Approach for Elastic Provisioning of Virtualized Cloud Resources". In: *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. 2011, pp. 45–54. DOI: 10.1109/MASCOTS.2011.47 (cit. on pp. 34, 35, 37, 38, 63, 64, 73, 77, 78, 84).

[117] Jia Rao et al. "DynaQoS: Model-free self-tuning fuzzy control of virtualized resources for QoS provisioning". In: *Quality of Service (IWQoS), 2011 IEEE 19th International Workshop on*. 2011, pp. 1–9. DOI: 10.1109/IWQOS.2011.5931341 (cit. on pp. 40, 41, 77).

[118] Jia Rao et al. "VCONF: A Reinforcement Learning Approach to Virtual Machines Autoconfiguration". In: *Proceedings of the 6th International Conference on Autonomic Computing*. ICAC '09. Barcelona, Spain: ACM, 2009, pp. 137–146. ISBN: 978-1-60558-564-2. DOI: 10.1145/1555228.1555263. URL: http://doi.acm.org/10.1145/1555228.1555263 (cit. on pp. 33, 37, 38, 44, 63, 64, 77, 87, 88).

[119] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 7:1–7:13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391236. URL: http://doi.acm.org/10.1145/2391229.2391236 (cit. on pp. 17, 23, 43, 49, 50, 94).

[120] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting". In: *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*. CLOUD '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 500–507. ISBN: 978-0-7695-4460-1. DOI: 10.1109/CLOUD.2011.42. URL: http://dx.doi.org/10.1109/CLOUD.2011.42 (cit. on p. 40).

[121] *RUBiS Online Auction System*. URL: http://rubis.ow2.org (visited on 09/17/2011) (cit. on pp. 54, 69, 79).

[122] Tudor-Ioan Salomie et al. "Application Level Ballooning for Efficient Server Consolidation". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 337–350. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465384. URL: http://doi.acm.org/10.1145/2465351.2465384 (cit. on p. 22).

[123] *Scalr cloud management*. URL: http://www.scalr.com/ (visited on 09/23/2014) (cit. on pp. 49, 51).

[124] Malte Schwarzkopf et al. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 351–364. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465386. URL: http://doi.acm.org/10.1145/2465351.2465386 (cit. on pp. 17, 23, 49).

[125] *Set up Autoscaling using Voting Tags*. URL: http://support.rightscale.com/12-Guides/Dashboard_Users_Guide/Manage/Arrays/Actions/Set_up_Autoscaling_using_Voting_Tags/index.html (visited on 04/12/2015) (cit. on pp. 35, 49, 51).

[126] Zhiming Shen et al. "CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems". In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 2011, 5:1–5:14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038921. URL: http://doi.acm.org/10.1145/2038916.2038921 (cit. on pp. 22, 31, 34, 43, 44, 51, 58, 73, 87, 88).

[127] Rob Sherwood et al. "Carving Research Slices out of Your Production Networks with OpenFlow". In: *SIGCOMM Comput. Commun. Rev.* 40.1 (Jan. 2010), pp. 129–130. ISSN: 0146-4833. DOI: 10.1145/1672308.1672333. URL: http://doi.acm.org/10.1145/1672308.1672333 (cit. on p. 21).

[128] Alan Shieh et al. "Sharing the Data Center Network". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 309–322. URL: http://dl.acm.org/citation.cfm?id=1972457.1972489 (cit. on p. 25).

[129] Bogdan Solomon et al. "A Real-time Adaptive Control of Autonomic Computing Environments". In: *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON '07. Richmond Hill, Ontario, Canada: IBM Corp., 2007, pp. 124–136. DOI: 10.1145/1321211.1321225. URL: http://dx.doi.org/10.1145/1321211.1321225 (cit. on p. 40).

[130] Basem Suleiman et al. "Trade-Off Analysis of Elasticity Approaches for Cloud-Based Business Applications". In: *Web Information Systems Engineering - WISE 2012 - 13th International Conference, Paphos, Cyprus, November 28-30, 2012. Proceedings*. Ed. by Xiaoyang Sean Wang et al. Vol. 7651. Lecture Notes in Computer Science. Springer, 2012, pp. 468–482. ISBN: 978-3-642-35062-7. DOI: 10.1007/978-3-642-35063-4_34. URL: http://dx.doi.org/10.1007/978-3-642-35063-4_34 (cit. on p. 35).

[131] Frezewd Lemma Tena, Thomas Knauth, and Christof Fetzer. "PowerCass: Energy Efficient, Consistent Hashing Based Storage for Micro Clouds Based Infrastructure". In: *Proceedings of the 2014 IEEE International Conference on Cloud Computing*. CLOUD '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 48–55. ISBN: 978-1-4799-5063-8. DOI: 10.1109/CLOUD.2014.17. URL: http://dx.doi.org/10.1109/CLOUD.2014.17 (cit. on p. 95).

[132] G. Tesauro et al. "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation". In: *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*. 2006, pp. 65–73. DOI: 10.1109/ICAC.2006.1662383 (cit. on p. 37).

[133] Gerald Tesauro et al. "On the use of hybrid reinforcement learning for autonomic resource allocation". In: *Cluster Computing* 10.3 (Sept. 2007), pp. 287–299. ISSN: 1386-7857. DOI: 10.1007/s10586-007-0035-6. URL: http://dx.doi.org/10.1007/s10586-007-0035-6 (cit. on p. 77).

[134] *The IRCache Project*. URL: http://www.ircache.net (visited on 09/17/2011) (cit. on pp. 54, 71, 86).

[135] B. Urgaonkar et al. "Dynamic Provisioning of Multi-tier Internet Applications". In: *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*. 2005, pp. 217–228. DOI: 10.1109/ICAC.2005.27 (cit. on p. 42).

[136] Bhuvan Urgaonkar et al. "Agile Dynamic Provisioning of Multi-tier Internet Applications". In: *ACM Trans. Auton. Adapt. Syst.* 3.1 (Mar. 2008), 1:1–1:39. ISSN: 1556-4665. DOI: 10.1145/1342171.1342172. URL: http://doi.acm.org/10.1145/1342171.1342172 (cit. on pp. 26, 42, 87, 88).

[137] Bhuvan Urgaonkar et al. "An analytical model for multi-tier internet services and its applications". In: *SIGMETRICS Perform. Eval. Rev.* 33.1 (June 2005), pp. 291–302. ISSN: 0163-5999. DOI: 10.1145/1071690.1064252. URL: http://doi.acm.org/10.1145/1071690.1064252 (cit. on p. 58).

[138] Luis Miguel Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. "Dynamically scaling applications in the cloud". In: *Computer Communication Review* 41.1 (2011), pp. 45–52. DOI: 10.1145/1925861.1925869. URL: http://doi.acm.org/10.1145/1925861.1925869 (cit. on p. 19).

[139] Venkatanathan Varadarajan et al. "Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense)". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 281–292. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382228. URL: http://doi.acm.org/10.1145/2382196.2382228 (cit. on pp. 21, 26).

[140] Nedeljko Vasić et al. "DejaVu: Accelerating Resource Allocation in Virtualized Environments". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 423–436. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2151021. URL: http://doi.acm.org/10.1145/2150976.2151021 (cit. on pp. 33, 77).

[141] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633 (cit. on pp. 94, 96).

[142] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments". In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ICAC '11. Karlsruhe, Germany: ACM, 2011, pp. 235–244. ISBN: 978-1-4503-0607-2. DOI: 10.1145/1998582.1998637. URL: http://doi.acm.org/10.1145/1998582.1998637 (cit. on p. 107).

[143] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015 (cit. on pp. 17, 22, 23, 25, 33, 49).

[144] Daniel Villela, Prashant Pradhan, and Dan Rubenstein. "Provisioning Servers in the Application Tier for e-Commerce Systems". In: *ACM Trans. Internet Technol.* 7.1 (Feb. 2007). ISSN: 1533-5399. DOI: 10.1145/1189740.1189747. URL: http://doi.acm.org/10.1145/1189740.1189747 (cit. on p. 42).

[145] Carl A. Waldspurger. "Memory Resource Management in VMware ESX Server". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 181–194. ISSN: 0163-5980. DOI: 10.1145/844128.844146. URL: http://doi.acm.org/10.1145/844128.844146 (cit. on p. 21).

[146] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. "The Xen-Blanket: Virtualize Once, Run Everywhere". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 113–126. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168849. URL: http://doi.acm.org/10.1145/2168836.2168849 (cit. on p. 20).

[147] Timothy Wood et al. "Black-box and Gray-box Strategies for Virtual Machine Migration". In: *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*. NSDI'07. Cambridge, MA: USENIX Association, 2007, pp. 17–17. URL: http://dl.acm.org/citation.cfm?id=1973430.1973447 (cit. on p. 20).

[148] Timothy Wood et al. "CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '11. Newport Beach, California, USA: ACM, 2011, pp. 121–132. ISBN: 978-1-4503-0687-4. DOI: 10.1145/1952682.1952699. URL: http://doi.acm.org/10.1145/1952682.1952699 (cit. on pp. 93, 95).

[149] Di Xie et al. "The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers". In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '12. Helsinki, Finland: ACM, 2012, pp. 199–210. ISBN: 978-1-4503-1419-0. DOI: 10.1145/2342356.2342397. URL: http://doi.acm.org/10.1145/2342356.2342397 (cit. on pp. 94, 107).

[150] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. "URL: A Unified Reinforcement Learning Approach for Autonomic Cloud Management". In: *J. Parallel Distrib. Comput.* 72.2 (Feb. 2012), pp. 95–105. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2011.10.003. URL: http://dx.doi.org/10.1016/j.jpdc.2011.10.003 (cit. on pp. 37, 38).

[151] Jing Xu et al. "On the Use of Fuzzy Modeling in Virtualized Data Center Management". In: *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*. 2007, pp. 25–25. DOI: 10.1109/ICAC.2007.28 (cit. on pp. 40, 41).

[152] Lenar Yazdanov and Christof Fetzer. "Lightweight Automatic Resource Scaling for Multi-tier Web Applications". In: *Proceedings of the 2014 IEEE International Conference on Cloud Computing*. CLOUD '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 466–473. ISBN: 978-1-4799-5063-8. DOI: 10.1109/CLOUD.2014.69. URL: http://dx.doi.org/10.1109/CLOUD.2014.69 (cit. on pp. 34, 75).

[153] Lenar Yazdanov and Christof Fetzer. "Vertical Scaling for Prioritized VMs Provisioning". In: *Proceedings of the 2012 Second International Conference on Cloud and Green Computing*. CGC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 118–125. ISBN: 978-0-7695-4864-7. DOI: 10.1109/CGC.2012.108. URL: http://dx.doi.org/10.1109/CGC.2012.108 (cit. on pp. 22, 25, 47).

[154] Lenar Yazdanov and Christof Fetzer. "VScaler: Autonomic Virtual Machine Scaling". In: *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 212–219. ISBN: 978-0-7695-5028-2. DOI: 10.1109/CLOUD.2013.142. URL: http://dx.doi.org/10.1109/CLOUD.2013.142 (cit. on pp. 33, 34, 61).

[155] Lenar Yazdanov, Maxim Gorbunov, and Christof Fetzer. "EHadoop: Network I/O Aware Scheduler for Elastic MapReduce Cluster". In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. 2015, pp. 821–828. DOI: 10.1109/CLOUD.2015.113 (cit. on p. 91).

[156] Matei Zaharia et al. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 265–278. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755940. URL: http://doi.acm.org/10.1145/1755913.1755940 (cit. on pp. 25, 102).

[157] Matei Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 29–42. URL: http://dl.acm.org/citation.cfm?id=1855741.1855744 (cit. on p. 77).

[158] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications". In: *Proceedings of the Fourth International Conference on Autonomic Computing*. ICAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–. ISBN: 0-7695-2779-5. DOI: 10.1109/ICAC.2007.1. URL: http://dx.doi.org/10.1109/ICAC.2007.1 (cit. on pp. 42, 58, 87, 88).

[159] Wei Zheng et al. "JustRunIt: Experiment-based Management of Virtualized Data Centers". In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. USENIX'09. San Diego, California: USENIX Association, 2009, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1855807.1855825 (cit. on pp. 33, 77).

[160] Xiaoyun Zhu et al. "What Does Control Theory Bring to Systems Research?" In: *SIGOPS Oper. Syst. Rev.* 43.1 (Jan. 2009), pp. 62–69. ISSN: 0163-5980. DOI: 10.1145/1496909.1496922. URL: http://doi.acm.org/10.1145/1496909.1496922 (cit. on pp. 39, 40).

LISTS OF FIGURES, TABLES AND ALGORITHMS

LIST OF FIGURES

LIST OF ALGORITHMS