Four-fold Auto-scaling on a Contemporary Deployment Platform using Docker Containers

Philipp Hoenisch[1,2], Ingo Weber[2,3], Stefan Schulte[1], Liming Zhu[2,3], and Alan Fekete[4,2]

[1]TU Wien, Austria, [2]Software Systems Research Group, NICTA, Sydney
[3]School of Computer Science & Engineering, University of New South Wales
[4]School of Information Technologies, University of Sydney, Australia
{p.hoenisch,s.schulte}@infosys.tuwien.ac.at,
<firstname>.<lastname>@nicta.com.au

Abstract. With the advent of Docker, it becomes popular to bundle Web applications (apps) and their libraries into lightweight linux containers and offer them to a wide public by deploying them in the cloud. Compared to previous approaches, like deploying apps in cloud-provided virtual machines (VMs), the use of containers allows faster start-up and less overhead. However, having containers inside VMs makes the decision about elastic scaling more flexible but also more complex. In this contemporary approach to service provisioning, four dimensions of scaling have to be considered: VMs and containers can be adjusted horizontally (changes in the number of instances) and vertically (changes in the computational resources available to instances). In this paper, we address this four-fold auto-scaling by formulating the scaling decision as a multi-objective optimization problem. We evaluate our approach with realistic apps, and show that using our approach we can reduce the average cost per request by about 20-28%.

1 Introduction

With the advent of Docker[1], lightweight containers are gaining wide-spread popularity among early adopter-type companies, including Facebook and Google [15]. Such containers are also particularly suitable to power recent trends like microservices and DevOps [1]. We consider the approach where a variety of different services or Web applications (apps) are running inside containers, each app deployed in instances of a particular container type. Various container instances are deployed on top of VM instances, which can be obtained from Infrastructure-as-a-Service (IaaS) providers, e.g., Amazon Web Services (AWS).

Virtualization brought the benefit of detaching the setup of a machine from a physical machine (PM), and enabled IaaS cloud offers. VMs contain a full operating system (OS) and all software required for a specific purpose. Further, a VM's configuration can be packaged into an image and cloned arbitrarily often. This mechanism is commonly used for apps running in the cloud [19]. Most recently, the idea of having an additional abstraction layer has been picked up

[1] http://docker.io, accessed 29/7/15

by lightweight containers which can be described as *smaller* variants of VMs: the OS is not included in a container; instead, the one from the host machine is used. However, most additional programs are included inside the container. Like VMs, containers offer *resource elasticity*, *isolation*, *flexibility* and *dependability*. Containers do need to run on compatible versions of the OS, and they share resources through the outside OS, but these limitations are compensated by benefits of faster start-up times (on the order of seconds, where booting a VM takes on the order of minutes) and less overhead in terms of used resources (since the containers do not require a separate OS). One can certainly use Docker containers (from now on only called *containers*)[2] *instead of* VMs directly on an OS that runs on private PMs. However, in public cloud platforms such as AWS, the norm is running containers *inside* VMs obtained from the IaaS providers. These VMs are initially launched with only the OS and no app-specific software. That raises a complicated question of allocation: *how many VMs are needed, and how should the containers be configured and distributed among the VMs?* Compared to instance allocation of VMs in former approaches, container deployment requires making allocation decisions from a much larger space of options. As in all cloud platforms, allocation needs to be dynamic and elastic: as the load on an application changes, the amount of resources used should be adjusted to match the change, so the costs can be scaled up and down following usage.

In this paper we propose to use *a multi-objective optimization model to solve the allocation problem for lightweight containers on top of VMs.* In this model, we consider horizontal and vertical auto-scaling on both the container and VM level. Hence, we make the following contributions:

- We offer a multi-objective optimization model for scaling a contemporary deployment platform (consisting of containers and VMs) including the interactions between the different layers. This has more degrees of freedom than scaling just VMs as in traditional cloud deployment scenarios.
- We show how the optimization model can be used for making auto-scaling decisions for this deployment platform. The control decisions require solving the optimization problem, which we can do for reasonably-sized distributed systems, even though the additional degrees of freedom make the optimization task more complex than for traditional approaches.
- Using a prototype, we provide a realistic evaluation of our approach against two baseline scenarios which consider fewer dimensions of scaling. In our experiments we achieved total savings of up to 28%.

The remainder of the paper is structured as follows. We start with a motivating example and give an overview of our approach in Section 2. In Section 3 we present the details of the optimization problem. We discuss the results of our experimental evaluation in Section 4 and related work in Section 5. Section 6 concludes the paper. An accompanying technical report (TR) [8] provides details that were omitted here for brevity.

[2] In our current implementation we use Docker as container technology. The approach however would work for any lightweight container supporting resource and application isolation, scalability and dependability.

2 Motivating Example

Consider the following a Platform-as-a-Service (PaaS) scenario: we assume the role of a provider hosting various apps using a public IaaS cloud service[3]. Apps have a specific *type* and are provided by different customers or content providers who want to have a fixed hosted solution. Each app may come with an optional *Service Level Agreement* (SLA), e.g., defining a maximal *response time* which should not be exceeded or a specific *throughput* which should be achieved. For the sake of simplicity, say the provider hosts the following three apps for three customers, all subject to varying workloads. *App A:* Joomla with extensions for appointment management, for customer *1*, a hairdresser; *App B:* Wordpress with plugins for CRM and Lead Management, for customer *2*, a tech startup; and *App C:* NodeJS with the web site of customer *3*, a skiing tour operator. As these apps would interfere with each other, each app is packaged into a separate container type. For that, a *dockerfile* specifies the configuration necessary to start containers, i.e., a new *container instance* of the same container type. New containers for the apps can then be instantiated as desired. Common practice is to specify version and build numbers or commit hashes in the dockerfile, so that all containers spawned from the same dockerfile start off as being the same, i.e., running the same application code, versions and libraries. If an app needs to be upgraded, this is initiated by changing the dockerfile.

The provider leases *VMs* of different *types*, i.e., each VM type has a different configuration in terms of supplied resources. In order to deploy containers onto a VM, a VM needs to be instantiated resulting in a VM *instance*. Notably, the provider can deploy many different containers (of different types) on a single VM instance. In addition, the provider may deploy a specific container type on several VM instances, resulting in various container instances where each container instance may have a different *configuration*. This configuration can be used to ensure the app's SLA is met, by defining requirements on the underlying VM, e.g., the resource demand in terms of CPU and RAM. The requirements on CPU are defined in *CPU shares*. By default, each VM has 1024 CPU shares available which are split between the hosted container instances. Say the provider leases a dual-core VM for all three apps and the respective numbers of requests rise during peak times – therefore the system needs to be scaled. The problem then is: *how many VMs of which types are needed, and how should the containers for the different apps be distributed among them, with which configuration?*

If the demand for all three apps increases more or less uniformly, it may be sufficient to lease more VMs, each hosting all three apps. However, if the demand for App A grows a lot faster than for App B and C, more resources need to be provisioned for it. If later App A's demand shrinks, the freed up resources may be released. Alternatively, if App B experiences increased demand at that point, the resources freed up by App A can be re-purposed for App B.

[3] Internal IT operation units face a very similar situation when providing container hosting to their organization. Hence, instead of a public cloud IaaS a private cloud can be used.

As can be seen, the decision of how many VMs of which type should be leased when and how to distribute and configure the containers is not straight forward. Hence, in this paper, we make use of an optimization approach and define four-fold auto-scaling as a multi-objective optimization model.

3 Optimization Approach

Building on the example scenario and the main aspects of our problem landscape, we now give an overview of our multi-objective optimization model. Due to space constraints, we present only the main objective function and refer to our technical report (TR) for more information [8]. The optimization model takes as input a set of different VM types ($V = \{1, ..., v^{\#}\}$) and container types ($D = \{1, ..., d^{\#}\}$). v corresponds to a VM type and k_v to a specific VM instance. Accordingly, d refers to a container type and c_d to a specific container instance (having a certain configuration). Each VM instance and container instance comprises a certain specification in terms of CPU and RAM. For the sake of simplicity, we generalize all types of resources here; in our implementation we differentiate CPU shares and RAM.

The goal of the objective function below is minimizing the overall cost, i.e., the cost for all leased VMs. Hence, the output of the model is two-fold: first it defines how many VM instances of which types are needed, and second, which container (including its configuration) should be deployed on which VM instance. The decision variable $x_{(c_d,k_v,t)}$ is set by the solver and defines which container c_d should be deployed on which VM instance k_v at time t.

$$\min \Bigg[\sum_{v \in V} c_v \cdot \gamma_{(v,t)} + \sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \big((1 - z_{(d,k_v,t)}) \cdot (x_{(c_d,k_v,t)} \cdot \Delta_d) \big) + \sum_{v \in V} \sum_{k_v \in K_v} \omega_f^R \cdot f_{(R,k_v,t)} + \sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \big(\omega_s \cdot s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)} \big) \Bigg]$$

The objective function comprises four terms. The first term $\sum_{v \in V} c_v \cdot \gamma_{(v,t)}$ computes the overall VM leasing cost: $\gamma_{(v,t)}$ many VM instances of type v with cost c_v are leased at time t. The second term $\sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \big((1 - z_{(d,k_v,t)}) \cdot (x_{(c_d,k_v,t)} \cdot \Delta_d) \big)$ sums up the time needed to deploy a container (Δ_d) on a VM instance k_v. If a specific container of type c_d gets deployed the first time on a VM instance k_v, some data needs to be downloaded from the container registry. Hence, this procedure may take some time. However, this data is cached on the VM instance as long as it is running, thus future deployments of the same container type will be much faster. In case a VM instance's cache contains already the needed data ($z_{(d,k_v,t)}$=1), the product inside the sums is 0. Further, this term prioritizes placement of containers on VM instances where they are cached already, since the term is minimized as part of the overall objective function. The third term $\sum_{v \in V} \sum_{k_v \in K_v} (\omega_f^R \cdot f_{(R,k_v,t)})$ computes the amount of *free* resources ($f_{(R,k_v,t)}$). This term ensures that containers are deployed on

already leased VM instances instead of leasing additional ones, provided enough resources are available. In order to control the contribution of this term towards the objective function, we weigh the free resources using the weight ω_f^R. The fourth term $\sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_s \cdot s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)}\right)$ sums up the amount of deployed containers for each container type at time t. It aims at avoiding over-provisioning on the container level by demanding to lease the smallest amount of resources to containers while still fulfilling the demand. As before, the term is weighed with a constant value ω_s.

The full optimization model can be found in the TR [8]. It includes numerous constraints, which define limits on valid container deployments and VM leasing plans. Overall, the optimization model ensures that enough resources are leased to handle the demand for each app at any time. Hence, the outcome of the optimization model is two-fold: 1) it determines whether additional VM instances need to be leased or if already leased VM instances can be terminated, and 2) it determines which container types to instantiate on which VM instance. Consequently, the system landscape subject to continuous change: VMs may appear or disappear at any time and containers may be moved between them.

4 Evaluation

Our optimization-based control of auto-scaling has been evaluated through measuring the behavior of a prototype. The source code is available at `http://reliableops.com`. Details of the architecture and evaluation can be found in the TR [8]. We compare our *optimized* approach against two state-of-the art baselines for making scaling decisions: *One-for-All* and *One-for-Each*. In both cases, additional resources are leased or released based on a threshold. *One-for-All* leases only quad-core VMs, where each VM hosts one container of each type. *One-for-Each* leases only single-core VMs, each hosting exactly one container. As in the example scenario in Section 2, we have three different container types (i.e., three apps). We test using two request arrival patterns, sending different and varying amounts of requests to each app.

Table 1: Evaluation Results

	Arrival Pattern 1			**Arrival Pattern 2**		
	Optimized	One-for -All	One-for -Each	Optimized	One-for -All	One-for -Each
Leased Cores	28.13	39.5	29.68	28.75	39.5	29.88
(σ)	(0.38)	(0)	(0.14)	(0.43)	(0)	(0.25)
Leasing Cost	378.4	505.6	474.67	393.2	505.6	478.00
(σ)	(0.92)	(0)	(2.31)	(1.38)	(0)	(4)
Cost/Invocations	0.17	0.23	0.21	0.17	0.24	0.22
(σ)	(0.012)	(0.22)	(0.02)	(0.03)	(0.06)	(0.04)
SLA Adherence	97.47%	98.02%	98.22%	96.95%	96.92%	97.1%

The result of our evaluation are shown in Table 1. As the numbers reveal, the SLA adherence for both arrival patterns and for each scaling strategy are very similar, varying by less than 2.5%. This can be expected given that we used the same thresholds for scaling up or down. Hence, in the following we focus on the leased CPU cores and incurred cost. It is not surprising that the *One-for-All* scenario produced the highest cost. In this scaling strategy, only quad-core VMs were leased, each hosting one container instance per app. This leads to a highly over-provisioned system when some apps experience relatively low load, as VMs are not be fully used in this case. Hence, leasing single-core VMs with only one app as in *One-for-Each* is ∼5% cheaper. However, even leasing smaller VMs may not be perfect. Since single-core VMs can not service as much load as more powerful VMs, more VMs are needed. In addition, in our cost model, leasing two single-core VMs is more expensive than leasing one dual-core VM (∼20% more expensive). Thus vertical scaling can be helpful: leasing the right VM size depending on the need may eventually lead to less leasing cost. Our four-fold optimization approach can take advantage of this, as can be seen in the *Cost/Invocations* row in Table 1: Eventually, for *Arrival Pattern 1*, we achieved savings of ∼28% (with ∼33% less cores) over the One-for-All scenario and monetary savings of ∼23% (∼4% less cores) over the One-for-Each. For *Arrival Pattern 2*, we achieved with our optimization cost savings of ∼25% (∼32% less cores) over the One-for-All scenario and savings of ∼20% (∼4% less cores) over the One-for-Each scenario.

5 Related Work

Resource allocation and auto-scaling are major research challenges in the field of cloud computing [3]. Several different approaches have been proposed for scaling single services, scientific workflows and business processes. Approaches for both fields differ in a number of aspects, e.g., process perspective, timeliness, resource insensitivity, scheduling on step-level or for the full process, etc. [6,7,16,17]. The major difference of process-based scaling versus scaling single services lays in the versatility, i.e., the amount of parallel requests may jump within a few seconds from a low to a very high number. Hence, the demand of resources may also change quickly which makes scaling decisions more complicated [2]. As resources are commonly paid for a fixed Billing Time Unit (BTU) (several minutes to a few hours), releasing resources before the end of such a cycle should be avoided.

The adherence to SLAs has also been investigated. SLAs are defined by customers under the objective of optimizing profit for the IaaS provider [11], or to achieve a high resource utilization [5,9,10]. For that, most approaches apply threshold-based scaling, i.e., fixed rules apply depending on the current load. Li et al. propose using reinforcement learning to reason about the best configuration to ensure a certain level of QoS [12]. However, it is more realistically for service providers to isolate different apps: they may have conflicts amongst each other, like exposing the same ports, using incompatible libraries, or privacy constraints preventing their deployment in the same VM [18].

To overcome this configuration problem, commonly apps and their libraries are packed into a single VM. Doing so, a more fine grained scaling is possible. However, from the point of view of a IaaS provider, the problem of unused resources has only been shifted. Now the question is how to use the PMs efficiently. A number of solutions have been proposed which consider different VM placement approaches with the aim of utilizing the physical resources efficiently [4,14,20]. As it is common that apps communicate with each other, having them in separate VMs will eventually lead to high data transfer. Cloud providers often charge for in and out-bound traffic. Hence, VM placement should consider a collocation of VMs which communicate regularly with each other [6,13]. Approaches for scaling VMs and placing them on PMs are particularly relevant to our work. However, using VMs instead of containers should be limited to cases where a full OS is needed while containers should be used to isolate single apps. Further, optimizing VM placement is only applicable for datacenter operators with hardware access. Where this is not the case, using containers in combination with VMs allows much finer-grained scaling and resource usage optimization.

6 Conclusions

The traditional approach of hosting apps directly on VMs suffers from several disadvantages, such as the overhead of a full OS and slow start-up time, a degree of vendor lock-in, and relatively coarse-grained units for scaling. In order to overcome those problems more and more organizations use lightweight container technologies like Docker. These add an additional abstraction layer on top of VMs, enabling more efficient use of VMs. When running containers on top of VMs, auto-scaling decisions have greater flexibility and greater complexity. In this paper, we defined a four-fold auto-scaling decision problem for this deployment scenario as a *multi-objective optimization model*, and we proposed a control architecture that dynamically and elastically adjusts VM and container provisioning. Based on a prototype implementation, which uses IBM CPLEX as optimization solver, we evaluated our approach extensively, comparing it against two naïve scaling strategies. We showed that our approach can choose and execute scaling decisions, achieving a cost reduction of 20-28% over the baselines. In our future work we want to extend this optimization model and consider the location of containers, privacy aspects, and long-running transactions.

Acknowledgments We thank An Binh Tran for sharing his technical expertise on Docker, and IBM for the academic license of CPLEX. NICTA is funded by the Australian Government as represented by the Department of Communications and the Australian Research Council through the ICT Centre of Excellence program. This work is partially supported by the European Union within the SIMPLI-CITY FP7-ICT project (Grant agreement no. 318201) and within the CREMA H2020-RIA project (Grant agreement no. 637066).

References

1. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional (2015)
2. Brousse, N.: Scaling on EC2 in a fast-paced environment. In: USENIX LISA (2011)
3. Buyya, R., Ranjan, R., Calheiros, R.N.: InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In: ICA3PP. LNCS, vol. 6081. Springer (2010)
4. Chen, W., Qiao, X., Wei, J., Huang, T.: A profit-aware virtual machine deployment optimization framework for cloud platform providers. In: IEEE CLOUD (2012)
5. Emeakaroha, V.C., Brandic, I., Maurer, M., Breskovic, I.: SLA-aware application deployment and resource allocation in clouds. In: COMPSAC Workshops (2011)
6. Hoenisch, P., Hochreiner, C., Schuller, D., Schulte, S., Mendling, J., Dustdar, S.: Cost-efficient scheduling of elastic processes in hybrid clouds. In: IEEE CLOUD (2015)
7. Hoenisch., P., Schuller, D., Schulte, S., Hochreiner, C., Dustdar, S.: Optimization of complex elastic processes. IEEE Transactions on Services Computing (2015)
8. Hoenisch, P., Weber, I., Schulte, S., Zhu, L., Fekete, A.: Four-fold auto-scaling for Docker containers. Tech. Rep. UNSW-CSE-TR-201513, University of New South Wales (2015), `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/201513.pdf`
9. Juhnke, E., Dörnemann, T., Bock, D., Freisleben, B.: Multi-objective scheduling of BPEL workflows in geographically distributed clouds. In: IEEE CLOUD (2011)
10. Kertesz, A., Kecskemeti, G., Brandic, I.: An interoperable and self-adaptive approach for SLA-based service virtualization in heterogeneous cloud environments. Future Generation Computer Systems 32, 54–68 (2012)
11. Lee, Y.C., Wang, C., Zomaya, A.Y., Zhou, B.B.: Profit-Driven Service Request Scheduling in Clouds. In: IEEE CCGrid (2010)
12. Li, H., Venugopal, S.: Using Reinforcement Learning for Controlling an Elastic Web Application Hosting Platform. In: IEEE ICAC 2011 (2011)
13. Meng, X., Pappas, V., Zhang, L.: Improving the scalability of data center networks with traffic-aware virtual machine placement. In: IEEE INFOCOM (2010)
14. Mills, K., Filliben, J., Dabrowski, C.: Comparing VM-placement algorithms for on-demand clouds. In: IEEE CloudCom (2011)
15. MSV, J., McCrory, C.: Is Docker a threat to the cloud ecosystem? (Aug 2014), `http://research.gigaom.com/2014/08/is-docker-a-threat-to-the-cloud-ecosystem/`, gigaom research
16. Schulte, S., Janiesch, C., Venugopal, S., Weber, I., Hoenisch, P.: Elastic business process management: State of the art and open challenges for BPM in the cloud. Future Generation Computer Systems 46, 36–50 (2015)
17. Schulte, S., Schuller, D., Hoenisch, P., Lampe, U., Dustdar, S., Steinmetz, R.: Cost-driven optimization of cloud resource allocation for elastic processes. International Journal of Cloud Computing 1(2), 1–14 (2013)
18. Shen, Z., Subbiah, S., Gu, X., Wilkes, J.: Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In: ACM SOCC (2011)
19. Varia, J.: Architecting for the cloud: Best practices. Amazon Web Services Whitepaper (2011)
20. Xu, J., Fortes, J.: Multi-objective virtual machine placement in virtualized data center environments. In: GreenCom - CPSCom (2010)