Improving Resource Efficiency of Container-instance Clusters on Clouds

Uchechukwu Awada and Adam Barker
School of Computer Science
University of St Andrews
St Andrews, Scotland, UK
Email: {ua5, adam.barker}@st-andrews.ac.uk

Abstract—**Cloud computing providers such as Amazon and Google have recently begun offering container-instances, which provide an efficient route to application deployment within a lightweight, isolated and well-defined execution environment. Cloud providers currently offer Container Service Platforms (CSPs), which orchestrate containerised applications.**

Existing CSP frameworks do not offer any form of intelligent resource scheduling: applications are usually scheduled individually, rather than taking a holistic view of all registered applications and available resources in the cloud. This can result in increased execution times for applications, resource wastage through underutilised container-instances, and a reduction in the number of applications that can be deployed, given the available resources.

The research presented in this paper aims to extend existing systems by adding a cloud-based Container Management Service (CMS) framework that offers increased deployment density, scalability and resource efficiency. CMS provides additional functionalities for orchestrating containerised applications by joint optimisation of sets of containerised applications, and resource pool in multiple (geographical distributed) cloud regions.

We evaluated CMS on a cloud-based CSP i.e., Amazon EC2 Container Management Service (ECS) and conducted extensive experiments using sets of CPU and Memory intensive containerised applications against the direct deployment strategy of Amazon ECS. The results show that CMS achieves up to 25% higher cluster utilisation, and up to 70% reduction in execution times.

I. Introduction

In order to fully exploit cloud computing [1] technologies organisations require a fast and efficient mechanism to deploy and manage complex applications across distributed resources. An increasing and diverse set of applications are now packaged in isolated user-space instances, on which they are executed. Such instances are called application or software containers. Application containers like Docker [2], wrap up a piece of software in a complete file-system, which contains everything it requires to run: code, runtime, system tools, system libraries etc. This packaging of an application also enables flexibility and portability on where the application can be executed e.g., on premises, cloud, bare metal, etc[1]. A single virtual machine can run several containers simultaneously. A recent analysis on Docker adoption in about 10,000 organisations[2] found that a typical Docker use case involves running five containers per host, but that many organisations run 10 or more.

[1]https://linux.com/news/docker-shipping-container-linux-code/
[2]https://datadoghq.com/docker-adoption/

Efficiently deploying and orchestrating containerised applications across distributed clouds is important to both developers and cloud providers. Cloud providers (i.e., AWS[3], Google Compute Engine[4]) currently offer Container Service Platforms (CSPs)[5,6], which support the flexible orchestration of containerised applications.

Existing CSP frameworks do not offer any form of intelligent resource scheduling: applications are usually scheduled individually, rather than taking a holistic view of all registered applications and available resources in the cloud. This can result in increased execution times for applications, and resource wastage through under utilised container-instances; but also a reduction in the number of applications that can be deployed, given the available resources. In addition, current CSP frameworks do not currently support: the deployment and scaling of containers across multiple regions at the same time; grouping containers into multi-container units in order to achieve higher cluster utilisation and reduced execution times.

This research aims to extend existing platforms by adding a cloud-based Container Management Service (CMS), which offers intelligent scheduling through the joint optimisation of sets of containerised applications across multiple cloud regions. Our aim is to maximize the overall Quality of Service (QoS) for containerised applications; in this paper we focus primarily on resource utilisation and execution time.

This paper makes the following contributions:

- **Capturing high-level resource requirements:** a representation which captures the high-level resource requirements of containerised applications along CPU, memory and network ports etc.
- **Efficient containers co-location:** techniques to group containers into multi-container (multi-task) units. This grouping serves as a unit of deployment on a container-instance, such that the aggregate resource requirement of a multi-container unit cannot exceed the total resources available on a container-instance.

[3]https://aws.amazon.com/
[4]https://cloud.google.com/compute/
[5]https://aws.amazon.com/ecs/
[6] https://cloud.google.com/container engine/

- **Optimal deployment:** a scheduling algorithm which solves the optimal deployment of sets of multi-container units on best fit container-instances across distributed clouds, such that to maximize all available resources, speed up completion time and maximise throughput [3], [4].

We evaluate CMS on Amazon's EC2 Container Management Service (ECS) platform across multi-region clusters in Oregon, Ireland, Frankfurt, North California, North Virginia, Tokyo, Singapore and Sydney. We conducted extensive experiments using sets of CPU and Memory intensive containerised applications against the direct deployment strategy of Amazon ECS.

The remainder is as follows. Section II discuss the problem formulation, Section III introduce CMS, Section IV presents the evaluation, Section V reviewed the related research and Section VI discuss the conclusion.

II. Theoretical Foundations

This paper considers resource and performance efficiency of containerised applications on a cloud provider. Resources are any cloud infrastructure components, such as CPU cores, Memory units, network communication ports etc or combination of these components in a container-instance. A container-instance cluster has one or more node instances (container-instance).

A. Problem Formulation

To connect with the joint optimisation with prior theoretical work, we cast into general formulations.

Notations: Given a set, $\mathbb{C}$ of containerised applications, each container serves as a task. A task $c \in \mathbb{C}$ can be divided into a collection of subtasks $\{(c, j)\}$. The jth subtask of the cth task has resource requirements along three resources: CPU, memory and network ports, as the total amount of resources needed for its execution, denoted as $d_{c,j}^{\langle c,m,p\rangle}$.

For each subtasks j in a task c, let t^s and t^c denote its start and completion times respectively. The execution time of the jth subtask is thus $t^e = t^c - t^s$. Finally, the aggregate execution time of a task is given as $\sum_{i=1}^{k} t_i^e / k$.

Given a cluster of container-instances $\mathbb{R}$ in a cloud region. Let $r^{\langle c,m,p\rangle}$ denote the resource capacity or available, in terms of CPU, memory and network ports respectively, of each container-instance $r \in \mathbb{R}$.

Next, we capture the resource demands $d_{c,j}^{\langle c,m,p\rangle}$ of n containerised application to be orchestrated , and get the update state of the cluster to obtain the resources available, $r^{\langle c,m,p\rangle}$. These information is important in order to make informed decision on orchestration. Next, we merge the tasks i.e., $\sum^{|r^{\langle c,m,p\rangle}|} \sum_j d_{c,j}^{\langle c,m,p\rangle}$ with capacity constraints to form new multi-tasks or multi-container units and deploy them to fully utilise available resources. A multi-container unit, is therefore a multi-task denoted as $\sum^{|r^{\langle c,m,p\rangle}|} \sum_j d_{c,j}^{\langle c,m,p\rangle} = d_{c,j}^{\langle c,m,p\rangle\prime}$. The aggregate execution time of a multi-container unit is given as $\sum_n \sum_{i=1}^{k} t_i^e / k = t^{e\prime}$

The resource utilisation of container-instances and the cluster is thus $\rho_{CI} = d_{c,j}^{\langle c,m,p\rangle\prime} / r^{\langle c,m,p\rangle}$, and $\rho_C = \sum_i d_{c,j}^{\langle c,m,p\rangle\prime} / \sum_i r_i^{\langle c,m,p\rangle}$ respectively.

Constraints: First, the cumulative resource requirements of a multi-container unit at any given time t cannot exceed the resource capacity or available of container-instances in the cluster:

$$d_{c,j}^{\langle c,m,p\rangle\prime} \leq r^{\langle c,m,p\rangle}, \forall_{c,m,p}. \tag{1}$$

Second, unused container-instance in the cluster would be shut down:

$$\forall_{c,m,p}\; r^{c,m,p} = 0 \text{ if } t \notin [t^s, t^c, t^e]. \tag{2}$$

Third, the utilisation of a cluster depends on application orchestration:

$$\rho_C = \max \left(\begin{array}{l} \sum^{|r^{\langle c,m,p\rangle}|} \sum_j d_{c,j}^{\langle c,m,p\rangle} = d_{c,j}^{\langle c,m,p\rangle\prime}\ \exists, \\ r^{\langle c,m,p\rangle} = 0 \text{ if } t \notin [t^s, t^c, t^e],\ \exists, \\ d_{c,j}^{\langle c,m,p\rangle\prime} \leq r^{\langle c,m,p\rangle}, \forall_{c,m,p}, \\ \forall_{c,m,p} \end{array} \right) \tag{3}$$

Forth, the overall execution times can be minimised depending on orchestration:

$$t^{e\prime} = \min \left(\begin{array}{l} \sum^{|r^{\langle c,m,p\rangle}|} \sum_j d_{c,j}^{\langle c,m,p\rangle} = d_{c,j}^{\langle c,m,p\rangle\prime}\ \exists, \\ d_{c,j}^{\langle c,m,p\rangle\prime} \to r^{\langle c,m,p\rangle} \\ \forall_{c,m,p} \end{array} \right) \tag{4}$$

In each term, $d_{c,j}^{\langle c,m,p\rangle\prime}$ is the total resource demand (e.g., CPU, memory, network ports) of a multi-task.

Objective: Our objective is to maximise the cluster resource utilisation and minimise the overall execution time of tasks. We denote the execution time of a multi-container unit as $t^{e\prime}$.

III. Container Management Service (CMS)

One of the current state-of-the-art Container Service Platforms (CSPs) is Amazon's EC2 Container Service (ECS). Amazon ECS, enables users to provision resources composed of container-instance clusters and deploy their containerised applications. First, users need to register these containers on the platform by providing a JavaScript Object Notation (JSON) definition, which is passed to the Docker daemon on a container-instance. The parameters in a container definition include: name, image, CPU and Memory demand, port-mappings, links etc. It uses the `Run Task` command to deploy each registered container randomly onto available container-instance that meets the parameters specified in its JSON definition.

This paper presents the Container Management System (CMS), which improves the current state-of-art in CSPs. The allocation of multiple tasks onto a machine to share the resources, can be used to increase resource utilisation [3], [5]. In CMS, we take into account set of containerised applications and resource pool management. Our algorithm reduces the

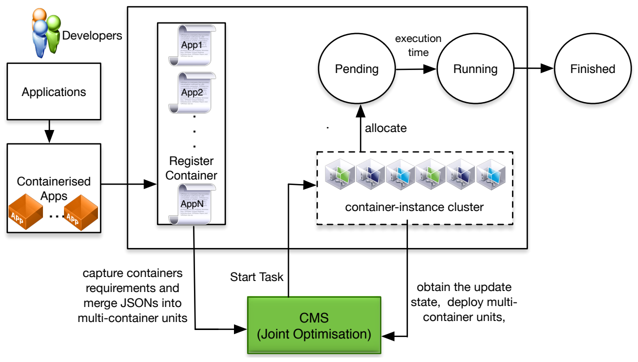

Developers
Applications
Containerised Apps
App1
App2
Register Container
AppN
Pending
execution time
Running
Finished
allocate
container-instance cluster
capture containers requirements and merge JSONs into multi-container units
Start Task
obtain the update state, deploy multi-container units,
CMS (Joint Optimisation)

Fig. 1: Orchestration overview of CMS: read from left to right.

number of required resources. We have realized our work on the Amazon EC2 Container Clusters provisioned at different geographical locations. Amazon's ECS cluster provides a simple solution to cluster management: region-specific and update state of cluster are exposed through API. The **`List`** and **`Describe`** API actions are used to find out what tasks are running and what instances are available in the cluster.

With reference to Figure 1, a basic flow through CMS is as follows:

- Preparation of application images
  - Developers use Docker to automate their applications into a container from a Dockerfile. These images are stored on-line in a container repository such as Docker Hub.
  - They are registered on the platform by providing a JavaScript Object Notation (JSON) definition.
- Execution of the applications
  - CMS captures the resource requirements of all containerised applications ready to be deployed at a period t.
  - CMS examines the cluster state to quantify the resource availability, such that we obtain detailed information of the available resources i.e., tasks (containers) running on the instances, resource availability or occupied (i.e., CPU, memory, network ports etc).
  - CMS deploys these containerised applications tightly onto available resources by merging their JSON definitions into multi-JSON definitions (multi-containers units) with constraints (Equation 3 and 4), such that resources are fully utilised and execution times are minimised.

All containers in a multi-JSON definition (multi-container unit) are deployed on the same container-instance and will be executed concurrently, thereby reducing the overall execution times. CMS uses the **`Start Task`** command to place these new multi-container units from a specified task definition, in a specified cluster and onto specified container-instances. This way, resources can be highly managed and utilised i.e., resources can be added or removed based on current demand.

IV. Experimental Evaluation

We evaluate CMS using sets of real life CPU and memory intensive applications in multi-container units with heterogeneous resource requirements across multi-region cloud clusters.

A. Setup

Clusters: On the large, we used 114 container-instances. A container instance has $1,024$ cpu units for every vCPU core and 995 units for every GiB of Memory. We provisioned clusters of Amazon container-optimized instances across the regions, each for our proposed system and Amazon ECS direct deployment strategy. We conducted extensive experiments and orchestrated sets of multi-container units with heterogeneous resource requirements across the regional clusters of **`t2.micro`** Intel Xeon Processors with Turbo up to 3.3GHz container-instances with $RegisteredResources$ = (1 vCPU, 1 vMem(GiB) and 5 Ports). The multi-region clusters configurations are shown on Table I.

Connection to Clusters: We have implemented our system on Amazon ECS clusters with boto3[7], the Amazon AWS SDK for Python. Boto3 is a data-driven and modern object-oriented API with consistent interface that supports JSON (JavaScript Object Notation) service definition. We import the boto3 module and create a connection to our Amazon ECS clusters.

Applications: To evaluate our framework, we illustrated use cases of real life CPU and memory intensive applications. The first application, denoted as app_1, is a memory intensive 3-tiered microservice application. This application consists of a simple frontend, an API and a redis backend. It is a simple statistics counter that increases every time a page is viewed[8], it runs three containers: frontend, the API and a redis container. The API communicates to a redis container to store data.

The second application, denoted as app_2, is a CPU/memory intensive word processor application and consists of 2 rake tasks. It takes in a message, posts a JSON message to a SQS queue, and polls the queue for messages and output them to standard output (stdout).

The third application, denoted as app_3, is wordpress[9] application. This a web software and a content management system based on PHP and MySQL. It communicates to a database name specified on MySQL container.

The last application, denoted as app_4, is nginx[10] application. This is an open source reverse proxy server for HTTP, HTTPS, SMTP, POP3, and IMAP protocols, as well as a load balancer, HTTP cache, and a web server.

B. Deployment Results

We discuss the detailed orchestration and results at each region, specifically focusing on CPU and Memory requirements.

[7]https://boto3.readthedocs.org/en/latest/

[8]http://blog.wercker.com/deploying-to-amazon-ec2-container-service-with-wercker

[9]https://hub.docker.com/wordpress/

[10]https://github.com/docker-library/docs/tree/master/nginx

TABLE I: Configuration of Multi-region clusters with aggregate CPU and Memory Capacities

Region	Clusters	CIs	Capacity, $r^{\langle c,m\rangle}$
`us-west-1`	CMS	8	$\langle 8192, 7960\rangle$
	Direct	8	$\langle 8192, 7960\rangle$
`ap-northeast-1`	CMS	6	$\langle 6144, 5970\rangle$
	Direct	6	$\langle 6144, 5970\rangle$
`us-east-1`	CMS	9	$\langle 9216, 8955\rangle$
	Direct	9	$\langle 9216, 8955\rangle$
`ap-southeast-1`	CMS	9	$\langle 9216, 8955\rangle$
	Direct	9	$\langle 9216, 8955\rangle$
`ap-southeast-2`	CMS	2	$\langle 2048, 1990\rangle$
	Direct	2	$\langle 2048, 1990\rangle$
`eu-central-1`	CMS	4	$\langle 4096, 3980\rangle$
	Direct	4	$\langle 4096, 3980\rangle$
`eu-west-1`	CMS	9	$\langle 9216, 8955\rangle$
	Direct	9	$\langle 9216, 8955\rangle$
`us-west-2`	CMS	10	$\langle 10240, 9950\rangle$
	Direct	10	$\langle 10240, 9950\rangle$

1) Resource Utilisation and Execution Time: Each experiment runs a combination of our applications (described above) merged into units. We see that CMS improves cluster resource efficiency, up to 25% and reduced execution times up to 70% compared to direct deployment.

In the **`us-west-1`** region, we deployed a set of 9 multi-container units, where each unit is running mixtures of $app1$, $app2$, $app3$, and $app4$. First, CMS captures the high-level resource demands of app_i specified in the JSON representation, gets update state and quantifies the resource availability at the regions (clusters), matches the resource demands and obtains a region having requisite capacity to accommodate the containers. At this period, **`us-west-1`** region has requisite capacity. Therefore, CMS merges the JSON representations such that $d_{c,j}^{\langle c,m,p\rangle\prime} \leq r^{\langle c,m,p\rangle}$ to form new set of (9) multi-container units, with resource requirements (CPU,Memory) $\langle 900, 900\rangle$, $\langle 1000, 850\rangle$, $\langle 1000, 900\rangle$, $\langle 950, 800\rangle$, $\langle 956, 812\rangle$, $\langle 950, 892\rangle$, $\langle 1000, 900\rangle$, $\langle 956, 912\rangle$ and $\langle 1000, 900\rangle$. CMS deploys these units onto best fit container-instances (i.e., $d_{c,j}^{\langle c,m,p\rangle\prime} \rightarrow r^{\langle c,m,p\rangle}$).
Figure 2a shows the details for the first experiment. CMS achieves higher utilisation (an average of 97%) when compared to the direct deployment (an average of 96%). The overall resource utilisation of CMS cluster is about 1% higher than the direct deployment.

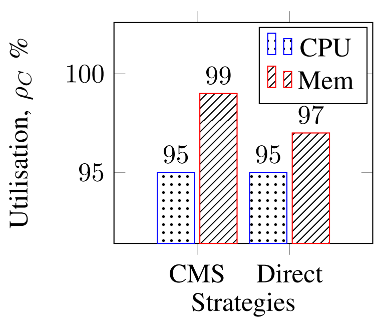
It is observed that the CMS method of merging JSON representations into deployable units is efficient as it not only allows for resources to be fully utilized at all times but also reduces the number of container-instances CIs needed for execution. Here we highlight that the gains are significant when compared to the direct deployment. In the **`us-west-1`** region, the execution times of applications is 3 times faster than applications deployed directly as shown in Figure 3.

In the **`ap-northeast-1`** region, we deployed sets of $app1$, $app2$, $app3$, and $app4$ and merged (multi-JSON) them into 5 multi-container units, such that $d_{c,j}^{\langle c,m,p\rangle\prime} \leq r^{\langle c,m,p\rangle}$ with each units resource requirements (CPU, Memory) $\langle 1000, 850\rangle$, $\langle 956, 812\rangle$, $\langle 1000, 900\rangle$, $\langle 950, 800\rangle$ and $\langle 900, 900\rangle$. Following the same procedure, CMS deployed these units onto 5 CIs and shut down the remaining unused CI in the cluster. Comparatively, we see in figure 2b that the direct deployment is unable to fully use available resources. CMS is better, it fully utilised available resources and shut down unused instance. Overall, CMS achieves about 14% (an average of 95%) higher utilisation, compared to the direct deployment (an average of 81%) and about 1.2 times faster execution, as shown in Figure 3.

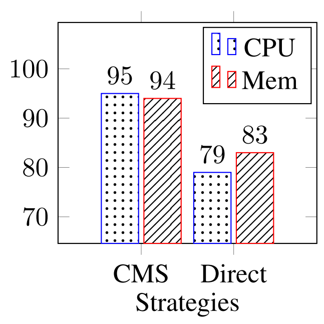
In the **`us-east-1`** region, we deployed sets of $app1$, $app2$, $app3$, and $app4$, merged (multi-JSON representations) into 7 multi-container units, such that $d_{c,j}^{\langle c,m,p\rangle\prime} \leq r^{\langle c,m,p\rangle}$ with diverse resource requirements (CPU, Memory) $\langle 950, 892\rangle$, $\langle 900, 900\rangle$, $\langle 1000, 900\rangle$, $\langle 1000, 850\rangle$, $\langle 956, 912\rangle$, $\langle 956, 912\rangle$ and $\langle 950, 800\rangle$. We observe that due to efficient packing, not only is the resources fully utilised at all times but also the reduction in the number of used resources. CMS deployed these units onto 7 container-instances CIs, and shut down free contain-instances in the cluster. It results to about 20% higher resource utilisation (an average of 95%) when compared to direct deployment (an average of 75%), as shown in Figure 2c. The execution time of CMS deployment is 2.2 times faster than direct deployment, as shown in Figure 3. Overall, CMS system performs better than direct deployment.

In the **`ap-southeast-1`** region, we deployed sets of $app1$, $app2$, $app3$, and $app4$, merged (multi-JSON representations) into 8 multi-container units, such that $d_{c,j}^{\langle c,m,p\rangle\prime} \leq r^{\langle c,m,p\rangle}$ with diverse resource requirements (CPU, Memmory) $\langle 950, 892\rangle$, $\langle 1000, 850\rangle$, $\langle 950, 912\rangle$, $\langle 950, 800\rangle$, $\langle 900, 900\rangle$, $\langle 956, 812\rangle$, $\langle 1000, 900\rangle$ and $\langle 1000, 900\rangle$. We summarize that avoiding under-allocation by explicitly packing on available resources improves efficiency. The gain in this experiment, are an increase in the number of simultaneously running applications, reduced overall application execution times, and improved usage of all cluster resources. These units are deployed into $8CIs$, shutting down the free CI in the cluster. The result in Figure 2d shows about 10% higher cluster utilisation for CMS deployment (an average of 97%) compared to the direct deployment (an average of 87%). In addtion, execution time is 2 times faster for CMS than direct deployment as shown in Figure 3.

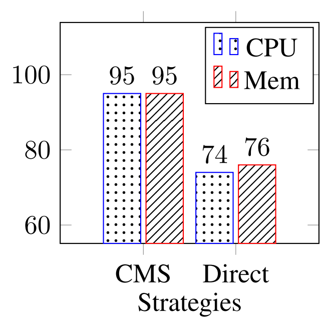
In the **`ap-southeast-2`** region, we deployed a set of 2 multi-container units consisting of $app1$, $app2$, $app3$ and $app4$ with diverse resource demands alone (CPU, Memory) $\langle 956, 912\rangle$ and $\langle 1000, 900\rangle$. CMS deployed these units onto $2CIs$. The result (Figure 2e) shows an average of 93% utilisation and faster execution time ($43s$) of all units compared to 93% utilisation of direct (cluster) deployment and execution time of $105s$. It is observed that there is no improvement in cluster utilisation. This is because CMS deployed the 2 multi-container units onto the 2 available container-instances in the cluster. In this case, there are no unused instances, as all instances are fully utilised in both deployments. However, the execution time is faster compared to direct deployment (2.4 times). Figure 3 shows the detailed execution times.



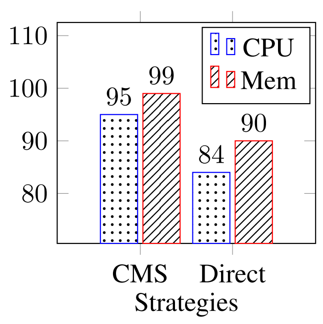
(a) At **us-west-1**



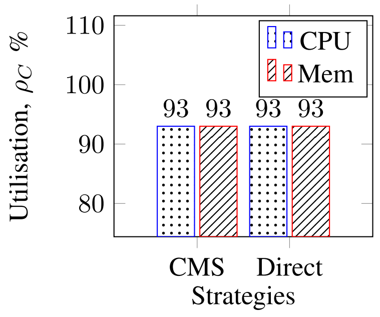
(b) At **ap-northeast-1**



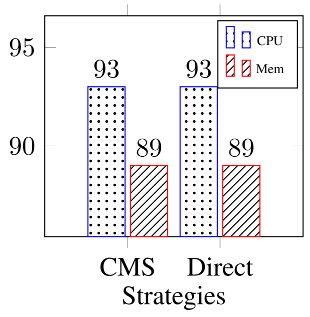
(c) At **us-east-1**



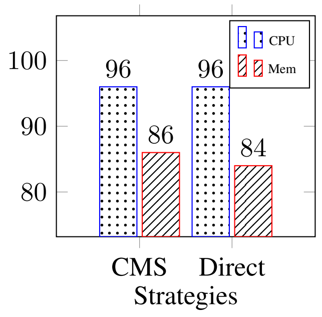
(d) At **ap-southeast-1**



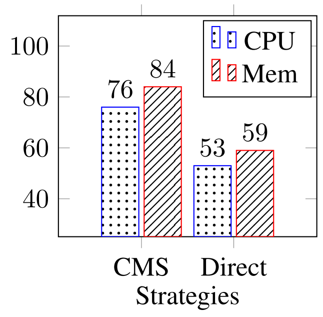
(e) At **ap-southeast-2**



(f) At **eu-central-1**



(g) At **eu-west-1**



(h) At **us-west-2**

Fig. 2: Resource utilisation at the multi-region clusters

In the **eu-central-1** region, we deployed a set of 4 multi-container units consisting of $app1$, $app2$, $app3$ and $app4$ with diverse resource demands $\langle 900, 900\rangle$, $\langle 950, 800\rangle$, $\langle 950, 892\rangle$ and $\langle 1000, 900\rangle$. Figure 2f compares the cluster utilisation of CMS with direct deployment methods. CMS achieves an average of 91% utilisation in the cluster and 91% utilisation in direct deployment method. We observed the same scenario in **ap-southeast-2** deployment. Resource are fully utilised in both deployments. However, the deployment of multi-container units guarantees faster execution. At this region, CMS achieves about 2 times faster execution of applications compared to direct deployment method. Figure 3 shows cluster wide execution times.

In the **eu-west-1** region, following CMS procedure, we deployed a set of 8 multi-container units from our applications ($app1$, $app2$, $app3$ and $app4$), with resource requirements (CPU, Memoey) $\langle 950, 892\rangle$, $\langle 900, 900\rangle$, $\langle 1000, 900\rangle$, $\langle 1000, 850\rangle$, $\langle 956, 912\rangle$, $\langle 956, 812\rangle$, $\langle 950, 800\rangle$ and $\langle 1000, 900\rangle$ onto $8CIs$. Figure 2g compares the cluster utilisation of CMS with direct deployment scheme. The result shows an average of 91% utilisation in CMS cluster and an average of 90% utilization in direct deployment cluster. CMS achieves about 1% higher utilisation. In addition, the execution time 1.8 times faster than direct deployment method, as shown in Figure 3.

In the **us-west-2** region, we deployed a set of 7 multi-container units consisting merges from JSON representations of $app1$, $app2$, $app3$ and $app4$, with resource requirements $\langle 950, 912\rangle$, $\langle 1000, 850\rangle$, $\langle 950, 892\rangle$, $\langle 950, 800\rangle$, $\langle 1000, 900\rangle$, $\langle 956, 812\rangle$ and $\langle 900, 900\rangle$. CMS deployed these units onto 7 CIs and shut down the remaining unused CI in the cluster. Resources are fully utilised in the CMS cluster and unused instances are shut down. CMS achieves higher cluster utilisation compared with direct deployment. It achieves an average of 80% utilisation with direct deployment of an average of 56% utilisation. Overall, CMS achieves about 24% higher utilisation, as shown in Figure 2h and 2 times faster execution time as shown in Figure 3.

2) Discussion: Overall, CMS has demonstrated superior QoS in container management and orchestration in cloud clusters. CMS deployment algorithm achieves higher cluster utilisation and minimized execution time of multi-container compared to direct Amazon ECS deployment strategy. We see that CMS contributes to up to 25% cluster utilisation and up to 70% reduced execution times. Increasing utilisation by a few percentage points can save millions of dollars in large scale computing [3].

Recall that the direct deployment method do not consider containerised application grouping and can under-allocate resource, causing a reduction in the number of applications that can be deployed. It also deploys application randomly on available resource without intelligent mapping. Hence, CMS gains are from avoiding under-allocation by capturing applications resource requirements, obtaining the update state of resource availability and carefully merging multiple JSON representations as a unit of deployment. These units are deployed to fully utilise available resource on container-instances. The gains from our experiments are as follows: an

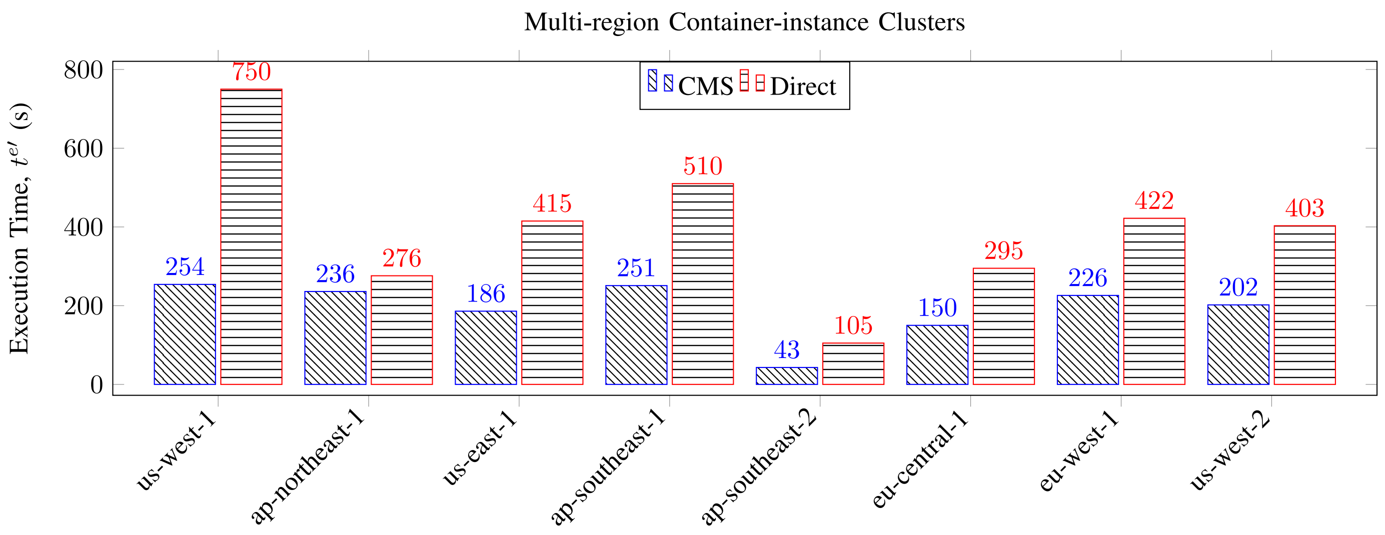


Fig. 3: Multi-containers execution times across multi-region clusters

increase in the number of applications that can be deployed at a time; reduced execution time of overall applications; improved usage of cluster resources.

V. Related Work

Recently launched cloud-based CSPs such as Amazon ECS, Google Container Engine etc, do not offer any form of intelligent resource scheduling: applications are usually scheduled individually, rather than taking a holistic view of all registered applications and available resources in the cloud. Throughout this paper we have demonstrated how this leads to lower utilisation, higher execution time, which in turn allow less applications to be deployed on a fixed set of resources. CMS has taken an early step towards meeting these challenges, by providing a framework for dynamic container orchestration. Resource management is an essential aspect of distributed systems. Reliable state management and flexible scheduling are essential in running modern distributed applications on clusters [6], [5], [3]. Academic and industrial researchers have developed several other cluster management frameworks for resource efficiency, such as Mesos [7], Omega [8], Borg [3] etc. Mesos [7] is a tool that abstracts and manages resources and scheduling in a cloud computing cluster. Omega [8] uses parallelism, shared state, and optimistic concurrency control, which is more advanced to [7]. Borg [3], is Google cluster manager that runs thousands of different applications across a number of clusters. It achieves high resource utilisation by packing different tasks together and assigning them to a machine if there are sufficient available resources that meet their constraints.

VI. Conclusion

This paper presented CMS, an approach for optimising containerised applications in multi-region cloud container-instance clusters. CMS takes into account the heterogeneous requirements of containerised applications, captures their high-level resource requirements, get an updated state of multi-region cloud clusters for resource availability, merges containers JSON representations to form new multi-container units and deploys these units on container-instances, such that higher throughput, high resource utilisation and faster execution times are achieved.

We have implemented CMS on Amazon EC2 Container Service (Amazon ECS) clusters and evaluated it against Amazon ECS direct deployment strategy. CMS has shown higher QoS (high resource utilisation, up to 25% and minimized execution time, up to 70%) compared to ECS direct deployment strategy.

References

[1] A. Barker, B. Varghese, J. S. Ward, and I. Sommerville, "Academic cloud computing research: Five pitfalls and five opportunities," in *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, USENIX Association, 2014.

[2] C. Anderson, "Docker [software engineering]," *IEEE Software*, vol. 32, pp. 102–c3, May 2015.

[3] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), pp. 18:1–18:17, ACM, 2015.

[4] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 589–603, USENIX Association, May 2015.

[5] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 455–466, Aug. 2014.

[6] C. Delimitrou and C. 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, (New York, NY, USA), pp. 127–144, ACM, 2014.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *NSDI*, vol. 11, pp. 22–22, 2011.

[8] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 351–364, ACM, 2013.