Resource Efficiency in Container-instance Clusters

Uchechukwu Awada
School of Computer Science
University of St Andrews
St Andrews, Scotland, UK
ua5@st-andrews.ac.uk

Adam Barker
School of Computer Science
University of St Andrews
St Andrews, Scotland, UK
adam.barker@st-andrews.ac.uk

ABSTRACT

Cloud computing providers 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 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, resource wastage through underutilized container-instances, and a reduction in the number of applications that can be deployed, given the available resources.

This paper presents a cloud-based Container Management Service (CMS) framework, which offers increased deployment density, scalability and resource efficiency for containerised applications. CMS extends the state-of-the-art by providing additional functionalities for orchestrating containerised applications by joint optimisation of sets of containerised applications and resource pool on the cloud. We evaluate 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 2.5 times faster execution times.

CCS CONCEPTS

•Computer systems organization → Distributed architectures; Cloud computing;

KEYWORDS

Application container, Cloud computing, Resource efficiency, Execution time

ACM Reference format:
Uchechukwu Awada and Adam Barker. 2017. Resource Efficiency in Container-instance Clusters. In *Proceedings of Second International Conference on Internet of Things and Cloud Computing, Cambridge, United Kingdom, March 2017 (ICC'17),* 5 pages.
DOI: http://dx.doi.org/10.1145/3018896.3056798

ICC'17, Cambridge, United Kingdom
 978-1-4503-4774-7/17/03…$15.00
DOI: http://dx.doi.org/10.1145/3018896.3056798

1 INTRODUCTION

In order to fully exploit cloud computing [2] technologies, organisations need a simple way to deploy, host and manage complex distributed applications across multiple 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 [1], wrap up a piece of software in a complete file-system that contain everything it needs to run: code, runtime, system tools, system libraries etc. A recent analysis on Docker adoption in about 10,000 organisations[1] found that a typical Docker use case involves running five containers per host, but that many organisations run 10 or more.

Efficiently deploying and orchestrating containerised applications on the cloud is important to both developers and cloud providers. Cloud providers (i.e., AWS[2], Google Compute Engine[3], etc.) currently offer Container Service Platforms (CSPs)[4, 5], which support the flexible orchestration of containerised applications. However, current systems 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 on 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.

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. 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 research contributions:

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

[1] https://datadoghq.com/docker-adoption/
[2] https://aws.amazon.com/
[3] https://cloud.google.com/compute/
[4] https://aws.amazon.com/ecs/
[5] 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, in order to maximize all available resources, speed up completion time and maximise throughput.

2 SYSTEM MODEL

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).

2.1 Problem Formulation

To connect with the joint optimisation problem 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:

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

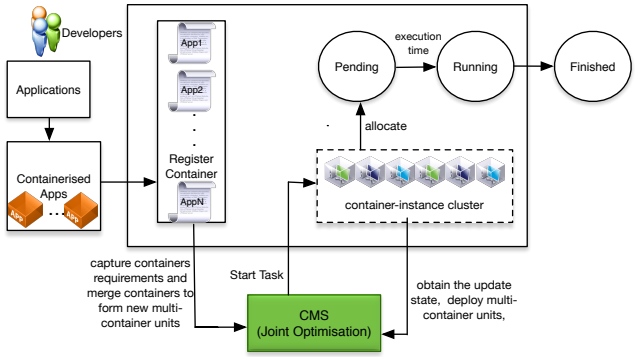

Figure 1: Orchestration overview of CMS: read from left to right.

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:

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

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 $t^{e\prime}$.

3 CMS SYSTEM

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 are 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 place each registered container randomly onto available container-instance in the cluster that meets the parameters specified in its JSON definition.

The approach taken by Amazon ECS and alternative CSPs can result in increased execution times for applications, and resource wastage through under utilised container-instances. This paper proposes the Container Management Service (CMS). As detailed in Figure 1, a basic flow is as follows:

Table 1: Cluster-wide Configuration

Clusters	Number of *CIs*	$\sum r^{\langle c,m,p \rangle}$ units
Experiment 1		
CMS	4	$\langle 4096, 3980 \rangle$
Direct	4	$\langle 4096, 3980 \rangle$
Experiment 2		
CMS	6	$\langle 6144, 5970 \rangle$
Direct	6	$\langle 6144, 5970 \rangle$
Experiment 3		
CMS	8	$\langle 8192, 7960 \rangle$
Direct	8	$\langle 8192, 7960 \rangle$
Experiment 4		
CMS	10	$\langle 10240, 9950 \rangle$
Direct	10	$\langle 10240, 9950 \rangle$

- 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[6].
  - 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 the available resources by merging them into a multi-containers units with capacity constraints, such that resources are fully utilised.

CMS uses the **`Start Task`** command to place these new multi-container unit 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

4 EXPERIMENTAL EVALUATION

We evaluate CMS using sets of different applications in multi-container units with heterogeneous resource requirements across cloud clusters.

4.1 Setup

Clusters: On the large, we used 56 container-instances of **`t2.micro`** Intel Xeon Processors with Turbo up to 3.3GHz container-instances with *RegisteredResources* = (1 vCPU, 1 vMem(GiB) and 5 Ports). A container instance has 1,024 cpu units for every vCPU core and 995 units for every GiB of Memory. The clusters configurations are shown on Table 1.

[6]https://hub.docker.com/

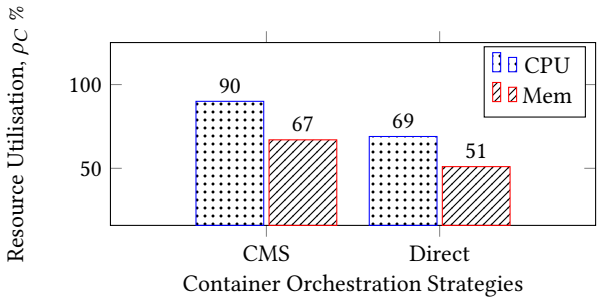

Figure 2: Resource utilisation for Exp 1 clusters

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 memory intensive 3-tiered microservice[8] application. This application consisted of a simple frontend, an API and a redis backend. It is a simple statistics counter that increases every time a page is viewed[9], 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 and post 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[10] 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[11] 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.

4.2 Deployment Results

Each experiment runs a combination of our applications ($app1$, $app2$, $app3$, and $app4$) merged into units. We see that CMS improves cluster resource efficiency, up to 25% and to 2.5 times faster execution time compared to direct deployment.

In Experiment 1, we deployed a set of applications, consisting of $app1$, $app2$, $app3$, and $app4$. First, CMS captures the high-level resource demands of each 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

[7]https://boto3.readthedocs.org/en/latest/
[8]https://en.wikipedia.org/wiki/Microservices
[9]http://blog.wercker.com/deploying-to-amazon-ec2-container-service-with-wercker
[10]https://hub.docker.com/wordpress/
[11]https://github.com/docker-library/docs/tree/master/nginx

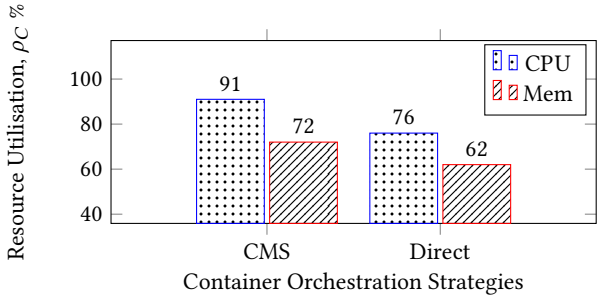

Figure 3: Resource utilisation for Exp 2 clusters

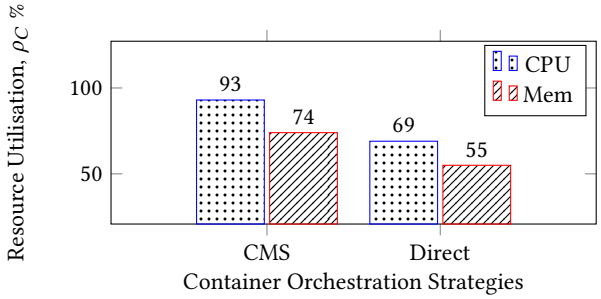

Figure 4: Resource utilisation for Exp 3 clusters

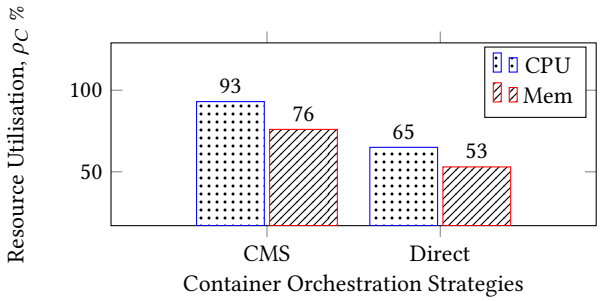

Figure 5: Resource utilisation for Exp 4 clusters

a container-instance having the requisite capacity to accommodate the merged unit. CMS merges the containerised applications, such that $d_{c,j}^{\langle c,m,p\rangle\prime} \leq r^{\langle c,m,p\rangle}$ to form new 3 multi-container units of resource requirements (CPU, Mem) $\langle 1000, 900\rangle$, $\langle 956, 812\rangle$ and $\langle 900, 900\rangle$. CMS deploys this unit onto best fit container-instance (i.e., $d_{c,j}^{\langle c,m,p\rangle\prime} \rightarrow r^{\langle c,m,p\rangle}$).

Figure 2 shows the details for the first experiment. First, CMS keeps a consistently higher number of running applications in the cluster; direct deployment, at all times, has scheduled fewer applications. CMS has higher throughput and usage of resources in the cluster. CMS achieves higher utilisation (an average of 78.5%) when compared to the direct deployment (an average of 60%). The overall resource utilisation of CMS cluster is about 19% higher than the direct deployment. CMS reduces overall execution times of containers deployed on cloud clusters. The execution time of

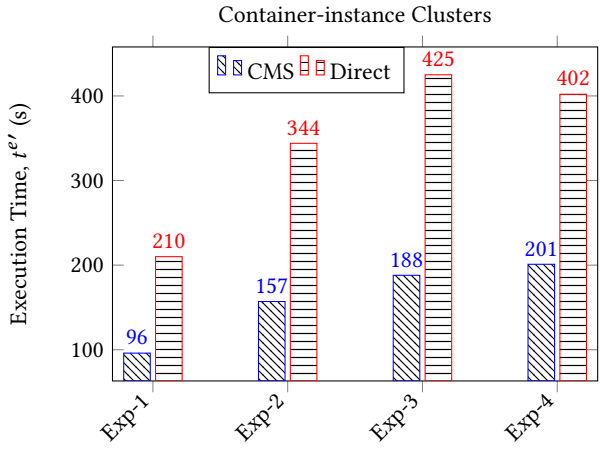

Figure 6: Execution times across the clusters

applications is 2.5 times faster than applications deployed directly as shown in Figure 6.

In Experiment 2, we deployed a set of applications, consisting of *app*1, *app*2, *app*3, and *app*4. CMS merged these into 5 multi-container units with diverse resource requirements (CPU, Mem) $\langle 950, 892\rangle$, $\langle 950, 800\rangle$, $\langle 1000, 900\rangle$, $\langle 956, 812\rangle$ and $\langle 900, 900\rangle$. We observe that due to efficient packing, not only are the resources fully utilised at all times, but also the number of resources are reduced. CMS deployed these units onto 5 container-instances *CIs*, and shut down free contain-instances in the cluster. This results in approximately 12.5% higher resource utilisation (an average of 81.5%) when compared to direct deployment (an average of 69%). This is shown in Figure 4. The execution time of CMS deployment is about 2.5 times faster than direct deployment, as illustrates in Figure 6.

In Experiment 3, we deployed 6 sets of multi-container units, consisting of *app*1, *app*2, *app*3, and *app*4, with resource requirements (CPU, Mem) $\langle 1000, 850\rangle$, $\langle 950, 892\rangle$, $\langle 950, 800\rangle$, $\langle 1000, 900\rangle$, $\langle 956, 812\rangle$ and $\langle 900, 900\rangle$. 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 83% utilisation with direct deployment of an average of 62% utilisation. Overall, CMS achieves about 21% higher utilisation, as shown in Figure 4 and about 2.5 times faster execution time as shown in Figure 6.

In Experiment 4, we deployed 7 set of a set multi-container units, consisting of *app*1, *app*2, *app*3, and *app*4, with resource requirements (CPU, Mem) $\langle 950, 912\rangle$, $\langle 1000, 850\rangle$, $\langle 950, 892\rangle$, $\langle 950, 800\rangle$, $\langle 1000, 900\rangle$, $\langle 956, 812\rangle$ and $\langle 900, 900\rangle$. Following the same procedure, CMS deployed these units and shut down the remaining unused *CIs* in the cluster. Comparatively, we see in figure 5 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 25.5% (an average of 84.5%) higher utilisation, compared to the direct deployment (an average of 59%) and about 200 seconds faster execution, as shown in Figure 6.

5 RELATED WORK

Cloud computing has attracted considerable attention as a technology for simplifying large scale computing. It has enhanced various organisation's computing by providing elastic virtual computing resources and platforms.

To further simplify these technologies, a new application packaging system that would guarantee fast deployment and execution was introduced. This new packaging system is refereed to as application container. Docker is a framework that automates the deployment of application inside containers, by providing an additional layer of abstraction and automation.

Recently launched cloud-based CSPs such as Amazon ECS, Google Container Engine, Apcera[12] etc, are orchestration systems for running containerised applications that are automated by Docker. These platforms enable users to set up clusters of container-instances and schedule their containerised applications into the clusters automatically based on requirements (such as CPU and Memory). Amazon ECS is customized for Docker containers and it provides API and scheduler to deploy containers on a managed cluster of container-instances. It allows to integrate a third-party schedulers to meet application specific requirements. Google Container Engine is built on the open source Kubernetes[13] system which allow the deployment and management of application containers in a cloud cluster. It allows finer control over containers, such as labelling and merging.

These 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. 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 [9], [6], [7], [3], [4], [10], [11]. Academic and industrial researchers have developed several other cluster management frameworks for resource efficiency, such as Mesos [5], Omega [8], Borg [11] etc. Mesos [5] is a tool that abstracts and manages resources and scheduling in a cloud computing cluster. It is based on two-level scheduling mechanism that shares resources e.g. CPU and memory in a fine-grained manner, which decides how many resources to offer each task, while tasks decide which resources to accept and which computations to run on them. Omega [8] uses parallelism, shared state, and optimistic concurrency control, which is more advanced to [5]. Borg [11], 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. It achieves this by: *feasibility checking*, to find machines on which the tasks could run, and *scoring*, which picks one of the feasible machines.

[12]https://www.apcera.com/
[13]http://kubernetes.io/

6 CONCLUSION AND FUTURE RESEARCH

This paper presented CMS, an approach for optimising containerised applications in cloud container-instance clusters. 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 (up to 25% high resource utilisation and 2.5 times faster execution times) compared to the ECS direct deployment strategy. We achieved this by first, capturing the resource requirements of each containerised application, second, we merge these containerised applications into multi-container units with capacity constraints and finally deploying them on best fit container-instances.

Modern academic experiments need high performance computing technologies to achieve optimal results. In addition, many academic experiments (i.e., medicine, physics, biology etc) generate big data which are geographical distributed. They require simplified technologies to help address such issue, many of whom have already adopted cloud computing technologies for easy and efficient processing of such big data. Our future research will explore more opportunities in academic experiments.

ACKNOWLEDGMENTS

This research is supported by the Amazon Web Services (AWS) Education Research Grant.

REFERENCES

[1] C. Anderson. 2015. Docker [Software engineering]. *IEEE Software* 32, 3 (May 2015), 102–c3.
[2] Adam Barker, Blesson Varghese, Jonathan Stuart Ward, and Ian Sommerville. 2014. Academic Cloud Computing Research: Five Pitfalls and Five Opportunities. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. USENIX Association.
[3] Christina Delimitrou and Christos Kozyrakis. 2014. 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)*. ACM, New York, NY, USA, 127–144.
[4] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466.
[5] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.
[6] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 4, 14 pages.
[7] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 36.
[8] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 351–364.
[9] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 35.
[10] A. Uchechukwu, K. Li, and K. Li. 2014. Scalable Analytic Models for Performance Efficiency in the Cloud. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. 998–1003.
[11] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 18, 17 pages.