2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

Towards Enabling Dynamic Resource Estimation and Correction for Improving Utilization in an Apache Mesos Cloud Environment

Gourav Rattihalli, Madhusudhan Govindaraju
Cloud and Big Data Lab
State University of New York at Binghamton
Binghamton, United States
{grattih1, mgovinda}@binghamton.edu

Devesh Tiwari
Department of Electrical and Computer Engineering
Northeastern University
Boston, United States
d.tiwari@northeastern.edu

***Abstract*—Academic cloud infrastructures require users to specify an estimate of their resource requirements. The resource usage for applications often depends on the input file sizes, parameters, optimization flags, and attributes, specified for each run. Incorrect estimation can result in low resource utilization of the entire infrastructure and long wait times for jobs in the queue. We have designed a *Resource Utilization based Migration* (RUMIG) system to address the resource estimation problem. We present the overall architecture of the two-stage elastic cluster design, the Apache Mesos-specific container migration system, and analyze the performance for several scientific workloads on three different cloud/cluster environments. In this paper we (b) present a design and implementation for container migration in a Mesos environment, (c) evaluate the effect of right-sizing and cluster elasticity on overall performance, (d) analyze different profiling intervals to determine the best fit, (e) determine the overhead of our profiling mechanism. Compared to the default use of Apache Mesos, in the best cases, RUMIG provides a gain of 65% in runtime (local cluster), 51% in CPU utilization in the Chameleon cloud, and 27% in memory utilization in the Jetstream cloud.**

I. Introduction

Computing resources consumed by applications can vary across runs, as the resource consumption is often dependent on configurations such as the input file sizes, optimization flags, input parameter choices, and the core application kernel. The current practice in academic/scientific cloud infrastructures such as Jetstream [1] and Chameleon [2], is that users are allocated a fixed number of Service Units (SU).

So, a user's estimate of required resources (CPU, Memory, GPU, I/O, Network bandwidth) and configuration of each node (or virtual machine) for the experiments is critical for requesting SUs and subsequent usage. In commercial clouds, the acquisition cost is dependent on the configuration of the virtual machines (VM) or bare metal nodes and the onus is on the user to be accurate. Therefore, it is critically important to have accurate resource request, and thereby, achieve high resource utilization, in both academic and commercial settings.

Resource allocation for an application is typically based on the estimate provided by the user. Therefore, incorrect estimation of resources by users can significantly increase the overall cost (or SUs) of running a set of applications [3]. For example, a snapshot of single day usage at SUNY Binghamton's Spiedie cluster, used exclusively by academic researchers, revealed that users tend to request a significantly higher amount of resources than what their applications require.

Over-allocation of resources for applications causes increased wait times for pending tasks in the queue, reduced throughput, and under-utilization of the cluster [4]. Cloud resource managers such as YARN [5], Omega [6], Mesos schedulers such as Apache Aurora [7] and Marathon [8], along with HPC resource managers such as Torque [9] and Slurm [10] – all require estimation for each application before it is launched. If the estimate is inaccurate, the cluster or cloud's overall utilization suffers.

Therefore, this paper explores new ways to augment the cluster management technologies so that they can more accurately estimate the resources requirements, in order to make the resource utilization more efficient and effective.

We have designed a Resource Utilization based Migration System (RUMIG) to address the resource estimation problem. The key idea behind RUMIG is to profile applications for a short period of time on a smaller-scale cluster to accurately estimate the resource requirements before migrating the application to the production cluster. RUMIG considers the user estimate as the starting point in the process of estimating the actual requirement. RUMIG uses an off-the-shelf Apache Mesos installation to demonstrate how RUMIG modules can be directly used by end users on academic clouds without the need for modifications to their existing installations.

RUMIG primarily focuses on workloads that may run on a single nodes on cloud computing platforms such as containerized jobs. RUMIG is aptly suitable for applications that do not have frequently varying phases and hence are a good candidate for RUMIG's initial profiling technique. For example, HPC and high-throughput applications that do not have varying phases can also benefit from RUMIG [11] [12] [13]. Recent studies [14], [15] have shown that a large fraction of jobs even on the cloud platform tend to run on single node (due to increase in microservices) and often utilize even lesser

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00033

resources than what a single node is capable – motivating the need for approaches such as RUMIG.

Specifically, in this paper, we make the following contributions:

- We present RUMIG - a two-stage elastic cluster design to profile an application at runtime in a Little Cluster and migrate it to an appropriate node in the Big Cluster.
- We evaluate and analyze the results on three different platforms: a local cluster (Arcus), Chameleon Cloud and Jetstream Cloud.
- We present analysis and discuss design trade-offs in enabling the container migration in Apache Mesos and how to apply this mechanism to improve resource utilization in a two-stage cluster.

In section II, we discuss the technologies we have used for the experiments. In section III, we present the design of RUMIG and the experimental setup on our local cluster (Arcus), Jetstream and Chameleon cloud platforms. In section IV, we present analysis of results on the local cluster (Arcus). In Section V, we present insights on applying RUMIG. In section VI, we compare the three platforms. In section VII, we discuss the potential of applying machine learning techniques. In section VIII, we present the related work and in section IX, we present future work. Finally, in section X, we present our conclusions.

II. Technologies used by RUMIG

In this section, we list the technologies and infrastructure used by our RUMIG design.

Mesos is a cluster-wide operting system that enables fine-grained resource sharing among co-scheduled applications.

Apache Aurora [7] is a widely used framework that is designed to submit and manage jobs to a Mesos cluster.

Docker is a well-known containerization platform. It uses the operating system (OS) on the host machine, unlike a Virtual Machine (VM) that has its own OS [16]. The container is an isolated environment and Docker virtualizes the system calls and routes them through the host OS, whereas a VM provides an abstraction at the hardware level. As a result, containers, such as Docker, are lightweight. The sharing of compute resources such as CPU and memory are enforced via *cgroups*.

Docker Swarm is a Docker native cluster manager that combines multiple Docker engines. It uses a *Manager-Agent* setup, where the manager assigns tasks to agents. We use the standalone version of Docker Swarm as it provides a centralized way to monitor resource utilization for each container.

Performance Co-Pilot is a lightweight Open Source toolkit for monitoring and managing system-level performance[17].

Checkpoint/Restore in Userspace (CRIU) is used for freezing a running application and checkpointing it. It can be further used to restore the application state. CRIU is currently being supported by Docker, but only in experimental mode.

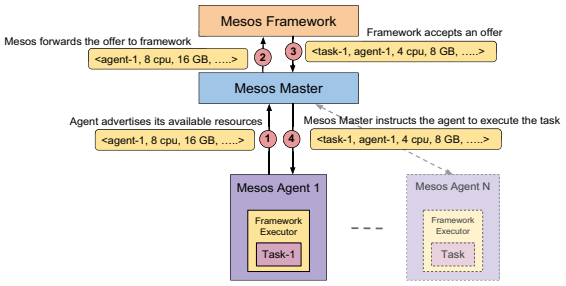


Figure 1. *Basic workflow of Mesos. The Mesos Framework used in our experiments is Apache Aurora.*

III. RUMIG: Design and Experimental Setup

A. Apache Mesos and Apache Aurora

Apache Mesos is used for infrastructure management and has a proven record of providing scalability for over thousands of nodes [18]. To enable efficient job scheduling in an Apache Mesos cluster, we chose the widely used Apache Aurora framework.

Mesos and Aurora. Figure 1 presents a basic workflow with Apache Mesos and Aurora. We use Apache Aurora as the Mesos framework. Mesos pools all the resources, i.e., number of CPU cores, amount of memory, disk space, and GPUs. In step 1, Mesos collects these compute resources across all Mesos agents/nodes as advertised by the agent (worker) nodes. In step 2, Mesos presents the offers to Aurora for its scheduling decisions. These offers contain a list of machines with their available free compute resources. An example of an offer from Mesos is the following:

[
<agent1-ID, cpu: 8, memory: 16GB,..>,
<agent2-ID, cpu: 6, memory: 6GB,..>,
...]

In step 3, Aurora picks an agent that is the best fit for the application to be launched. Aurora by default follows a First-Fit scheduling policy. In step 4, Mesos receives the acceptance of an offer from Aurora and instructs the agent specified by Aurora to launch the application. In this setup, a job can be scheduled just on a single agent. As a result, the application can execute just on a single machine. HPC jobs that require multiple agents/nodes cannot currently be launched by this setup.

Next, Figure 2 shows how an application's resources are estimated, migrated and launched. From a given allocation of nodes/VMs, we logically separate the nodes into a Big and Little cluster. To estimate the usage of resources for a given application, we begin execution in the Little Cluster. The applications launched in the Little Cluster get the user requested resources. Resource usage for the application is measured at one-second frequency until the standard deviation for the last 5 seconds is less than 5% of the average for the same duration. The resources measured are CPU and memory. Based on the collected data, we then estimate the amount that needs to be allocated for each resource using the following formula:

$$buffer = \left| \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \right|$$

$$Required\ Resource = Median\ Of\ Observations + buffer$$

Equation 1. *In the formula, N denotes the total number of observations, x_i denotes the ith observation and $\overline{x}$ denotes the arithmetic mean of the observations.*

The buffer is a positive deviation from the observed resource utilization and we consider the required resource to be the sum of the median of all observation and the buffer. The buffer is required to provide some headroom for the application. It is critical for Aurora-Mesos as the application will be terminated by Mesos if the application uses more than the allocated resources.

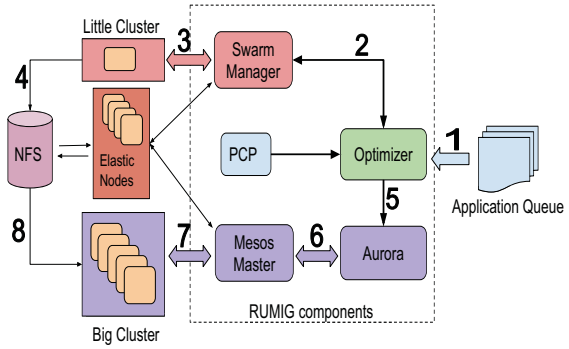


Figure 2. *Architecture of RUMIG. The figure shows the steps for profiling, checkpointing and resuming in the cluster. The figure also shows how the elastic nodes are setup as part of both the Swarm and Mesos clusters.*

The Big Cluster is used for further execution of the application with resource allocation based on the estimation in the Little Cluster. In the first step, the application is added to the job queue in the optimizer. The optimizer sends the application to the Docker swarm manager. Next, the optimizer starts monitoring the application and when the estimates for the application are ready, the optimizer creates a checkpoint of the application and creates a Docker image of the container file system. Both of these are stored in the NFS, which is shared across all the agents of the two clusters. Then, the optimizer sends the application with the estimated resources, the location of the checkpoint and the Docker image to the Mesos master. The Mesos master allocates an agent to the application. Once the application is received by the agent, the checkpoint data is loaded by CRIU and the application is resumed on the agent.

RUMIG-base is a simplified version of RUMIG that does not involve container migration and elastic nodes. Instead, it restarts the container after profiling, which is similar to Kubernetes VPA for cases that require only one restart. Hence, RUMIG-base provides a fair comparison basis with Kubernetes VPA.

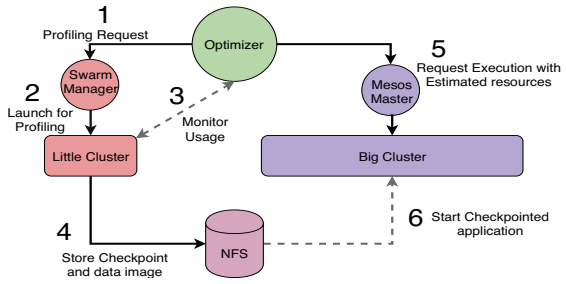


Figure 3. *Container Migration workflow. The figure shows the steps involved in profiling and migrating an application.*

B. Container Migration for Mesos

RUMIG consists of two clusters, one where the profiling takes place and the other where the application executes to completion. The containers are migrated from one cluster to another to continue execution after profiling. This process requires integration of container migration in RUMIG. Container Migration is not supported by Mesos, and so we designed and implemented it as a module in RUMIG.

Apache Mesos has no direct support for container migration and does not have the full capability to execute Docker containers yet. It can just launch and monitor a Docker container. Mesos decides where the job will be launched based on its scheduling policies. This means that until the job is launched there is no way to know where the checkpointing data, Docker image, and temporary files generated by the container need to be moved. So, efficient migration of containers is a challenge in Mesos clusters.

The CRIU functionality, though available in Docker, cannot be used in Docker's standard mode and is available only in *Experimental* mode.

In Figure 3, we explain the steps involved in container migration. The optimizer asks the Swarm manager to launch the task for the purpose of profiling (step 1). Swarm manager launches the application on the Little cluster (step 2). The application on the Little Cluster gets the user requested amount of resources. The optimizer now starts monitoring the application using Performance Co-Pilot (PCP) (step 3). Once the resources are estimated, the optimizer creates a checkpoint and a Docker image that contains the data generated by the application. The checkpoint and the Docker image are stored in NFS (step 4). Next, the optimizer asks the Mesos master to launch the application with the estimated resources, the checkpoint, and the Docker image with the application generated data (step 5). It is to be noted that Mesos cannot migrate containers, but it can launch Docker containers. We migrate the application as a new Docker container that contains the application generated data. The checkpoint is passed as a volume to the container. When the application launches, the Docker image, and the checkpoint is pulled from the NFS and the application resumes execution (step 6).

Using the steps described in Figure 3, we have added the ability to migrate the containers from the Little Cluster to the Big Cluster. The current design also has an elastic cluster setup - when the load on one of the clusters is high, we move

nodes from the other cluster. All the nodes in our setup are agents of the Mesos cluster, while a fixed number of nodes are agents of the swarm cluster. So, when a node is temporarily part of both clusters and is switched to the swarm cluster, that node is added to the maintenance mode on the Mesos cluster. Maintenance nodes do not accept jobs from the Mesos master.

C. System Setup

We tested our approach on multiple environments: (1) Arcus Cluster, which is our local cluster; (2) VMs on Jetstream cloud, and (3) VMs on Chameleon cloud. Table I, provides details of the hardware, Operating System and other software used in the experimental setup on Arcus Cluster, Jetstream cluster and Chameleon Cloud cluster.

Equipment/OS/Software	Description/Version
Arcus Nodes (15) - Bare metal nodes	Intel Xeon E5345, 8 core processor at 2.3GHz, 16 GB DDR2 RAM
Jetstream Nodes (9) - Virtual Machines	Intel Xeon E5-2680 v3, 6 core processor at 2.5GHz, 16 GB DDR3 RAM
Chameleon Nodes (11) - Virtual Machines	Intel Xeon E312xx v3, 8 core processor at 2.3GHz, 16 GB DDR3 RAM
Operating System	Ubuntu 16.04
Docker	17.06.1-ce
Apache Mesos	1.5.1
Apache Aurora	0.17.0

Table I. *Description of the infrastructure used in the experiments on Arcus, Jetstream and Chameleon Cloud clusters.*

D. Benchmarks/Workloads

To test the applicability of our solution we applied it to several workloads as indicated below. We have included applications with wide range of characteristics to demonstrate the applicability of our approach to a wide range of cloud applications.

1) SPECjvm2008: This is a Java Virtual Machine benchmark, that measures the performance of a Java Runtime Environment (JRE) and contains several real-world applications that focus on core Java functionality. The benchmark suite includes scientific applications such as Monte-Carlo simulations, that are used for various applications from biology to quantum physics [19]. The applications of Fast Fourier Transform, included in this suite, are well known in several areas including spectral analysis and signal processing.

2) DGEMM: It is a simple, dense, multi-threaded matrix multiply application and measures a sustained, floating-point computational rate of a node.

3) Princeton Application Repository for Shared-Memory Computers (PARSEC): It is a benchmark suite consisting of diverse multi-threaded applications ranging from media processing to financial application. We have used PARSEC 3.0 in our experiments [20], [21], [22].

Workload	Description
1. Blackscholes	Computational financial analysis application
2. Canneal	Engineering application
3. Ferret	Similarity search application
4. Fluidanimate	Application consists of animation tasks
5. Freqmine	Data mining application
6. Swaptions	Financial Analysis application
7. Streamcluster	Data mining application
8. DGEMM	Dense-matrix multiply benchmark
9. SPECjvm2008	A suite containing floating point benchmarks (Monte-Carlo, FFT, LU, SOR and Sparse)

Table II. *Description of the benchmarks from PARSEC (1-7), DGEMM and SPECjvm2008 used in the experiments.*

E. Elasticity of Little and Big Cluster Sizes

We ran multiple experiments with different setup sizes for the Big and Little cluster setup to determine the ideal ratio between the size of the Little and Big Cluster. The ideal ratio needs to be determined for each infrastructure setup and workloads separately. For all the experiments, we also discuss the results of container migration in the Big-Little setup and compare them with the results of both the default Mesos and RUMIG-base setups. In all the experiments that use the default Mesos setup, we have labeled the graphs with the format DM: *number of nodes*. The experiments with RUMIG-base are labeled with the format *(setup-ratio) RUMIG-base*. Note that the default Mesos and RUMIG setups present only agent nodes. They always have one extra master node. The RUMIG-base ratio includes the master node. The ratios are mentioned in the format *Little-Cluster-size:Big-Cluster-size*. Initially, RUMIG includes all elastic nodes in the Little cluster. Algorithm 1 is used to make a decision about moving the elastic nodes from the Little cluster to the Big cluster and vice versa. We configured the setup to assign one elastic node in the Little cluster for every 10 jobs in the queue (1:10 ratio). So, if the job queue decreases to affect the 1:10 ratio, an elastic node is moved from the Little cluster to the Big cluster. If the job queue size increases, an elastic node is moved back to the Little cluster. Each time an elastic node with the least number of running jobs is selected to move. As a job may still be running on that node, it is first added to the Mesos maintenance mode list. Mesos does not schedule new tasks on maintenance nodes. Once the running job on the node completes, the node is registered with the Little or Big Cluster, as appropriate. The Little cluster always has a minimum of one node.

The experiments use of a mix of 90 benchmarks, 10 instances of each of the benchmarks presented in Table II. The benchmarks were randomly shuffled for each run, as the arrival rate and order of applications may affect the results. All the presented data are average of 10 runs.

F. Profiling Interval Analysis

To determine the interval to be used for profiling, we carried out experiments at various intervals. These intervals are 3, 5, and 7 seconds. Tables III and IV, show the results of profiling

Algorithm 1: RUMIG: Cluster Elasticity

```
1  OptimalSize = ceil(queue/10);
2  if LittleClusterSize ≠ OptimalSize then
3      if LittleClusterSize < OptimalSize then
4          TargetNode = select a node that is part of
             ElasticNodes list & belongs to the BigCluster &
             has the least number of running jobs;
5          MaintenanceMode.add(TargetNode);
6          wait();// wait for applications to
               finish execution
7          Register node on LittleCluster;
8      else
9          TargetNode = select a node that is part of
             ElasticNodes list & belongs to the LittleCluster
             & has the least number of running jobs;
10         wait();// wait for applications to
               finish execution
11         MaintenanceMode.remove(TargetNode);
12         Deregister node on LittleCluster;
13     end
14 end
```

with these intervals. At 3 second intervals, the results are not satisfactory. We observe that there some instances where the approximations are incorrect by almost a factor of two. For example, in SPECjvm2008 we observe a memory approximation of 1805.21 MB whereas the actual requirement was 3350 MB. For the CPU approximations, the actual requirement is 5 cores whereas the profiler returned 2 cores. The memory approximation for Canneal is also incorrect. However, at 5 and 7 second interval we see that the profiler is much more accurate. Except for Freqmine, where the 5-second interval takes 10 seconds to right-size. We determined that the 5-second interval is a good trade-off when compared to the 7-second interval – the 7-second interval's accuracy is similar to the 5-second interval, but on average it takes longer to right-size an application.

IV. Results on Arcus Cluster

1) RUMIG vs Kubernetes VPA: Kubernetes provides a Vertical scaling system called Vertical Pod Autoscaler (VPA), which aims to provide the required amount of resources to a container running inside a Kubernetes pod. This is accomplished by monitoring, estimating and restarting the containers. If the initial resource estimate given to Kubernetes is not within the 90th percentile, the VPA keeps re-starting the application with estimates based on the following formula: *New Resources = max(peak+MinBumpUp, peak*BumpUpRatio)* where, *MinBumpUp = 100MB/m & BumpUpRatio = 1.2*, till the vertical scaling meets the resource requirement ('m' is the CPU unit used by Kubernetes, 1000m = 1 cpu core). In the worst case, the estimation system in Kubernetes can also exceed the maximum available resources on the nodes causing the pod to go a pending state [23]. Our RUMIG-base is similar to VPA because it profiles and then re-starts the application once. Figure 4 compares RUMIG-base with Kubernetes VPA, running in default namespace. Kubernetes VPA does multiple restarts to reach the appropriate resources required to run the container. The ideal case is when the user provides near optimal resources for the tasks so that VPA requires just one restart. So, we provided optimal+30% extra resources required, to run the container. RUMIG-base, which does not use checkpoint/restore, performs similar to the best case for Kubernetes VPA. RUMIG, however performs significantly better for cases in which Kubernetes requires multiple restarts. as it eliminates the need for restarts.

Workload	Profiling Interval			Actual
	3 seconds	5 seconds	7 seconds	Reqmt
Blackscholes	574.87 (6)	683.83 (5)	670.10 (7)	649
Canneal	540.14 (3)	935.94 (5)	985.86 (7)	945.1
Ferret	106.01 (3)	105.49 (5)	105.53 (7)	105
Fluidanimate	503.09 (3)	511.02 (5)	511.09 (7)	511
Freqmine	352.03 (3)	596.65 (10)	583.19 (7)	601
Swaptions	6.18 (3)	6.44 (5)	6.52 (7)	6.5
Streamcluster	110.44 (3)	110.91 (5)	110.66 (7)	110
Dgemm	25.61 (3)	25.65 (5)	25.65 (7)	24.9
SPECjvm2008	1805.21 (6)	3287.66 (15)	3290.83 (14)	3350

Table III. *Table shows the results of the memory profiling methodology along with the actual memory requirements of each application as determined by static profiling. The values in parenthesis show the number of seconds taken for each approximation. (Units: MB). While most approximations are good at 3 seconds, the 5 and 7 second readings more closely meet our accuracy requirements.*

Workload	Profiling Interval			Actual
	3 seconds	5 seconds	7 seconds	Reqmt
Blackscholes	2 (6)	2 (5)	2 (5)	2
Canneal	1 (3)	1 (5)	1 (7)	1
Ferret	1 (3)	1 (5)	1 (7)	1
Fluidanimate	2 (3)	2 (5)	2 (7)	2
Freqmine	1 (3)	1 (10)	1 (7)	1
Swaptions	3 (3)	3 (5)	3 (7)	3
Streamcluster	3 (3)	3 (5)	3 (7)	3
Dgemm	6 (3)	7 (5)	7 (7)	7
SPECjvm2008	2 (6)	5 (15)	5 (14)	5

Table IV. *Table shows the results of profiling to determine the number of cores, along with the actual requirements of each application as determined by static profiling. The values in parenthesis show the number of seconds taken for each approximation. (Units: CPU cores). While most approximations are good at 3 seconds, it sometimes fails to achieve the accuracy we require, as seen in Blackscholes (longer time) and SPECjvm2008 (incorrect approximation).*

In the rest of the experiments we do not compare RUMIG with VPA, as the two have different approaches - VPA makes incremental resource allocation changes in each iteration, along with a restart and so has a huge overhead compared to the RUMIG. We instead study the overhead and performance of RUMIG for various setups.

2) Equal Cluster Sizes: 7:7 setup: The Figure 5a shows the runtime of all the setups. With the default Mesos setup,

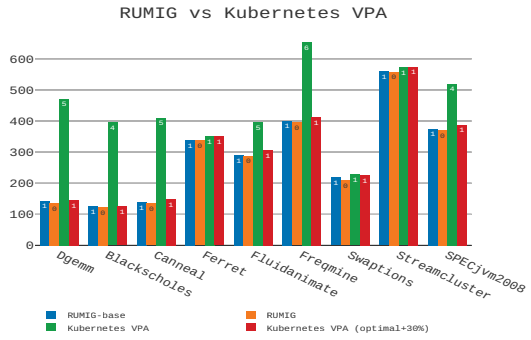


Figure 4. *Runtime comparison between RUMIG-Base case, RUMIG, Kubernetes VPA and Kubernetes VPA with pods assigned optimal+30% resources. The number on the top of each bar indicates how many restarts were required for the application to get adequate resources.*

we get a runtime of 3319 seconds to execute the workloads. The RUMIG-base approach has a runtime of 1282 seconds (includes time spent for profiling) – an improvement of 61%. This RUMIG-base experiment was conducted with a setup of 1 node in the Little Cluster for 14 nodes on the Big Cluster. We subsequently ran RUMIG with a configuration that starts with 50% distribution of nodes each on the Big and Little cluster. At 7:7 setup, RUMIG takes an average of 1290 seconds, which is within 1% of the RUMIG-base approach. The difference can be attributed to the overhead of moving nodes from the Big Cluster to the Little Cluster, which results in a queue on the Big Cluster where the real execution takes place. We also observe lower memory utilization by 5% (see Figure 5c) and an improvement in CPU utilization by 6% when compared to RUMIG-base (see Figure 5b).

3) 5:9 setup: In the next experiment, we reduced the size of the Little Cluster so that the ratio with the Big Cluster is 5:9. We observe an improvement in the runtime to 1187 seconds, a gain of about 8% over 7:7. We also see an increase in memory utilization when compared to the 7:7 setup by about 5%. However, the memory utilization is similar to the RUMIG-base approach. The CPU utilization shows an improvement of 11% when compared to the RUMIG-base approach. This improvement in runtime is due to the higher number of nodes in the Big Cluster.

4) 3:11 setup: In order to determine the optimal ratio of the cluster sizes, we further reduced the size of the Little Cluster to a 3:11 setup. At this ratio, we see a significant improvement in the throughput. The overall runtime dropped from 1282 in RUMIG-base to 1134 seconds on average using RUMIG with the 3:11 setup. Compared to the default Mesos setup, there is a significant difference of 2185 seconds, which is 65% improvement. We also see gains with CPU and memory utilization. The memory utilization improves by about 5% and CPU utilization by about 17% compared to RUMIG-base. The CPU utilization is enhanced by about 84% compared to the default Mesos setup. Due to the reduced runtime, and the ability to pack more jobs on each node, makes the system more efficient.

V. Additional Analysis of RUMIG

1) Overhead of Right-sizing the resource requirements: In Figure 7a, we compare the time taken to determine the correct amount of resources for all setups. Each experiment has the same number of workloads in the queue. We observe that the 7:7 setup is the quickest to rightsize the entire workload, which takes about 21 seconds. The 5:9 setup takes longer at 29 seconds and 3:11 setup is the slowest at 40 seconds. Even though the 3:11 setup is slowest among the three, it is still approximately three times faster than the RUMIG-base approach. We can conclude that 3:11 setup has the best overall results, even though 7:7 has the best time to determine the resource requirements. The 7:7 setup tends to make the applications wait in the Big Cluster, which reduces the overall cluster resource utilization. The 5:9 setup efficiently determines the resource requirements, but only gives a marginal improvement in CPU and memory utilization. The 3:11 setup provides efficient time for right-sizing at 40 seconds while significantly improving the CPU usage by 84% and memory utilization by 7% when compared with the default Mesos setup.

2) Cluster Elasticity Demonstration: In Figure 6a, we show an example of the elasticity of our Big Cluster compared to the behavior of the default Mesos setup. The results shown in the figure only contain the first few seconds of the execution. The first few seconds is ideal to show the elasticity as the cluster size grows and is shown by the increasing cluster CPU utilization, which corresponds to an increase the overall number of nodes in use. The experiment is set up with 7 nodes in the Little Cluster and 7 nodes in the Big cluster. We use the DGEMM benchmark, a CPU intensive workload. We compare against the default Mesos setup with the same workload. We can observe from Figure 6a that the default Mesos is quick to start executing the applications, while our Big-Little setup slowly increases the Big Cluster in size and eventually has a much higher average CPU utilization. In this example, 6 nodes move from the Little Cluster to the Big Cluster. Note that if a node is scheduled to be moved to another cluster, no applications are further scheduled on that node and the node is moved when all the running applications have completed executing on the node.

3) Effects of Migration: In Figure 6b, we present the runtime of the applications for different execution methods. These three methods have the following configuration: (1) use of *standalone Docker*, where we get the runtime of the application without the use of Aurora-Mesos; (2) the runtime without migration, but with the use of Aurora-Mesos; and (3) the runtime with migration and the use of Mesos, which is also the RUMIG setup with no elastic nodes. In all the runs we see that the standalone Docker performs the best as expected, as it does not have the scheduling overhead that is incurred by Aurora and Mesos. When we use the Aurora-Mesos setup, we observe an average increase in runtime by about 4% and with migration the increase in runtime is about 3%. We observe that on average the RUMIG performs better by about 2% over default Mesos. In some cases, the migration system is much

slower as seen in Ferret, Fluidanimate, Streamcluster, and SPECjvm2008. The reason for the higher runtime on Ferret, Fluidanimate and Streamcluster is that they generate a lot of data by the time the checkpoint is created. This data is stored in the image. To restart the application on the target node, there is an overhead cost for migration of the data. The SPECjvm2008 benchmarks is another example where migration is expensive. As the base image of SPECjvm2008 itself is very large, it adds a significant overhead in the overall time to migrate. It is to be noted that SPECjvm2008 is a suite of benchmarks in the same image and hence the larger size.

4) Downtime: In Figure 6c, we show the average downtime of applications. In the standalone mode, there is no downtime as the applications start as soon as they are scheduled. In the case of RUMIG-base or without migration, the downtime is the time spent for profiling in the Little Cluster. As the application is restarted on the Big Cluster after profiling. In the case of RUMIG or with migration, the application execution in the Little Cluster is not considered as downtime. This is because the application will be migrated to the Big Cluster and will continue execution. The figure shows that the downtime for applications is much less with RUMIG than with RUMIG-base. However, with applications that require a lot of data transfer such as Ferret, Fluidanimate, Streamcluster, and SPECjvm2008, the downtimes are expectedly higher in RUMIG.

5) Comparing Allocation vs Usage: Figures 7b and 7c, show the cluster-wide CPU and memory allocation and usage for the default Mesos and RUMIG setup with elastic nodes. The figures show the effectiveness of our profiling and container migration modules.

VI. RUMIG - Comparison across Arcus, Jetstream, and Chameleon

In Table V and VI, we show the cluster wide performance results for different setups and compare them with the default use of Mesos and RUMIG-base. We observe improvements in the cluster utilization regardless of the type of infrastructure.

The Table VII presents the comparison of results obtained in the three platforms. The Arcus cluster has bare metal nodes, while Jetstream and Chameleon have Virtual Machines. The infrastructure provided by both Jetstream and Chameleon are on shared nodes, where multiple users may have virtual machines on the same node. Also, the nodes are of a different processor class – Jetstream nodes have 6 cores each and Chameleon nodes have 8 cores each. The nodes in all three setups have the same amount of memory. The memory and processor technologies are not uniform across the three platforms. Chameleon node configurations are similar to Arcus. The best ratio in our experiments for Arcus is 3:11, Jetstream is 2:6, and for chameleon it is 1:9. While there are differences in the magnitude of the gains, RUMIG does provide improvements on all three platforms compared to the default Mesos setup.

Setup	Runtime	CPU Utilization	Memory Utilization
DM: 8 nodes	2374	38.14%	55.91%
1:8 RUMIG Base	901	58.30%	68.98%
4:4 RUMIG	926	56.84%	60.83%
3:5 RUMIG	847	69.67%	66.83%
2:6 RUMIG	779	75.10%	70.81%

Table V. *Results on Jetstream Cluster.*

Setup	Runtime	CPU Utilization	Memory Utilization
DM: 10 nodes	2498	28.1%	47.9%
1:10 RUMIG Base	2272	72.44%	54.21%
5:5 RUMIG	1899	73.88%	53.44%
3:7 RUMIG	1839	77.97%	55.60%
1:9 RUMIG	1631	78.89%	58.37%

Table VI. *Results on Chameleon Cluster.*

	Arcus	Jetstream	Chameleon
Type	Bare metal	Virtual Machine	Virtual Machine
Configuration - Each node	CPU - 8 cores, Mem - 16 GB	CPU - 6 cores, Mem - 16 GB	CPU - 8 core, Mem - 16 GB
Best Ratio	3:11	2:6	1:9
CPU Utilization	76.43%	75.10%	78.89%
Memory Utilization	64.31%	70.81%	58.37%

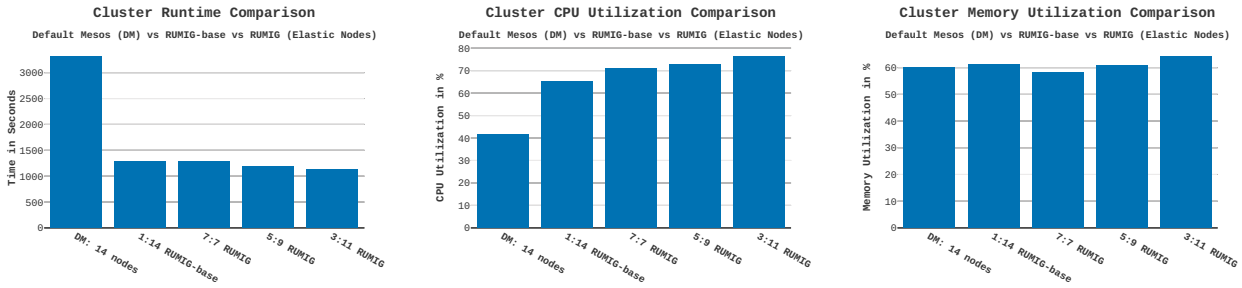
Table VII. *Comparison of results obtained with different platforms. The table shows that RUMIG provides improvements regardless of the platform and configuration.*

VII. Potential of Classification Techniques

We explored the potential for classification techniques to improve the time taken to estimate the resource requirement of an application. For example, if an application has been previously recorded with resource estimation then the application can be launched with the previously recorded data, or with the predicted values even if the input configurations and target infrastructure are changed. Additionally, we could classify the users who submit the workloads into categories depending on their past history of how accurate they were with the initial estimates. To study the potential of this approach, we conducted some experiments on Arcus cluster.

- We carried out a set of experiments where we assumed the accuracy of the machine learning technique for classification to be 33%, 66%, and 100%. Figure 8a shows the results for run-time. We see that at 100% accuracy, the results are expectedly promising and comparable to the 3:11 configuration.
- At 66% accuracy, the runtime is 9% higher, the CPU utilization is down by 7% and memory utilization is down by 5% when compared to 3:11 configuration.
- At 100% accuracy, the runtime is 7% lower, the CPU utilization is 2% higher and memory utilization is 3% higher when compared to 3:11 configuration. Note that 100% accuracy refers to cases when an application's resource requirements are known as they are submitted with the same input parameters as their previous run(s).

These results show that a combination of RUMIG and machine learning techniques could bring improvements in the time required to optimize the resource requests.

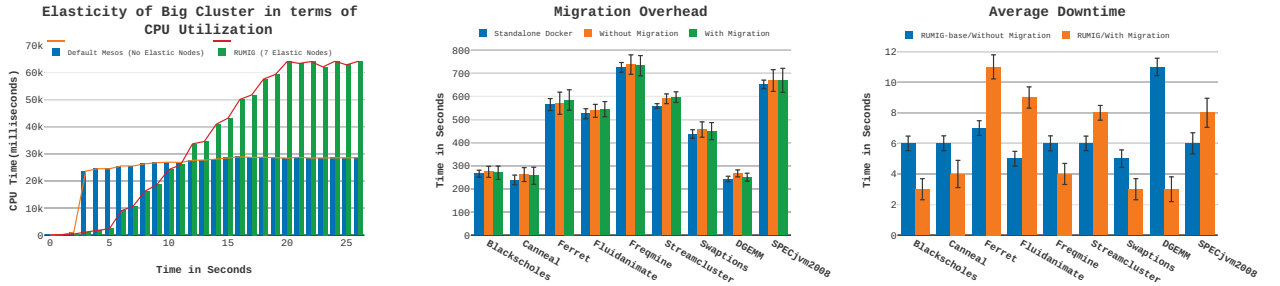


(a) *Runtime comparison of all setups on Arcus cluster. The 3:11 setup shows the best runtime followed by 5:9 setup.*

(b) *CPU utilization of all setups on Arcus cluster. The 3:11 shows the best CPU utilization followed by 5:9 setup.*

(c) *Memory utilization of all setups on Arcus cluster. The 3:11 setup shows the best memory utilization followed by 1:14 RUMIG-base.*

Figure 5. *Runtime, CPU usage and Memory usage comparison on Arcus Cluster.*

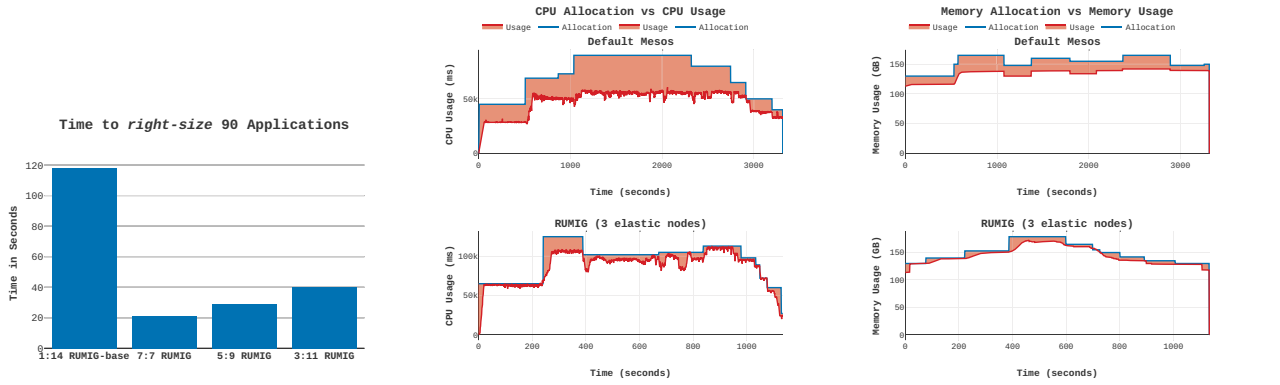


(a) *Elasticity of Big cluster in terms of CPU utilization on 7:7 setup and the default Mesos setup. The graph shows how the the number of nodes in the Big cluster grows (Elasticity) with the increase in CPU utilization. The green bars show the elastic growth in number of nodes, while the blue bars represent the default Mesos nodes.*

(b) *Effects of migration on different benchmarks. Ferret, Streamcluster and SPECjvm2008 show that the migration overhead can cause slowdown.*

(c) *Average downtime of different benchmarks. The downtime includes the cost entire migration and the time Mesos takes to schedule on the Big cluster. The error bars are determined over 10 runs.*

Figure 6. *Elasticity, Migration Overhead and Average downtime comparison on Arcus Cluster.*

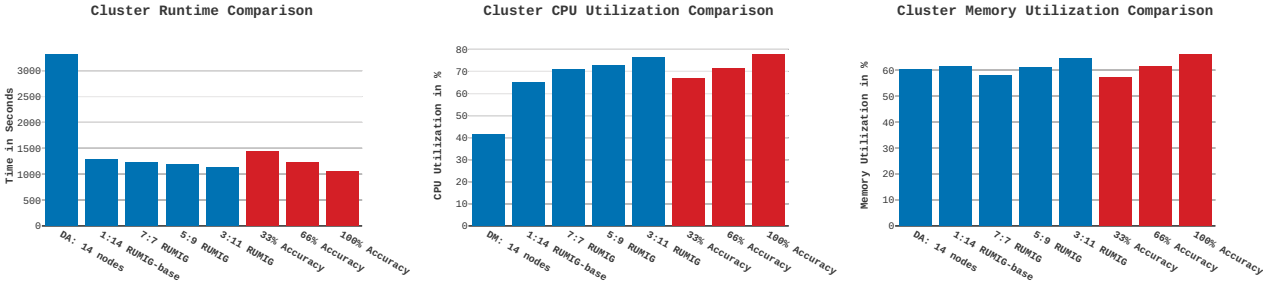


(a) *Total time spent by 90 applications on the Little cluster in different setups. The 7:7 setup is the fastest to right-size all the applications due to higher availability of elastic nodes.*

(b) *Comparison between CPU allocation and CPU usage on default mesos and RUMIG with 3 elastic nodes.*

(c) between Memory allocation and Memory usage on default mesos and RUMIG with 3 elastic nodes.

Figure 7. *Right-sizing time, CPU and Memory allocation vs usage comparison on Arcus Cluster.*



(a) *Runtime comparison of Default Mesos, RUMIG-base, all RUMIG setups and the setups configured using potential Machine Learning based classification on Arcus Cluster.*

(b) *CPU utilization comparison of Default Mesos, RUMIG-base, all RUMIG setups and the setups configured using potential Machine Learning based classification on Arcus Cluster.*

(c) *Memory utilization comparison of Default Mesos, RUMIG-base, all RUMIG setups and the setups configured using potential Machine Learning based classification on Arcus Cluster.*

Figure 8. *Possible effects of using Classification techniques on Runtime, CPU usage and Memory usage in Arcus Cluster.*

VIII. Related Work

In our previous work, TSRO, we explored the possibility of dynamically profiling applications [24]. In another previous work, we listed the challenges in implementing a container migration system in Apache Mesos [25]. This paper significantly extends our previous work. In TSRO, the application has to be restarted on the Big Cluster once the profiling is completed. The TSRO approach negatively affects the run-time and results in wastage of resources, i.e. results from the partial execution on the Little Cluster are discarded. RUMIG supports container migration in Mesos to address this inefficiency. TSRO also does not work for applications that provide a service as it has to be terminated in the Little Cluster, which affects the quality of service. Unlike RUMIG, TSRO does not support elasticity and the size of the Little and Big Clusters is fixed.

Hoenisch et al. [26] developed a framework for vertical and horizontal scaling of containers by restarting the applications. Our work assumes that continuous execution is important, and restarting is not acceptable.

Dhuraibi et al. [27] summarized the challenges involved in container elasticity and include resource availability as a key factor. RUMIG addresses several of the challenges listed by Dhuraibi et al.

Baresi et al. [28] presented a vertical scaling capability for web applications at a VM and container level. Hadley et al. [29] have shown the capability of CRIU (Checkpoint Restore In Userspace), a linux utility for live container migration, across multiple clouds. They show this capability for both stateful and stateless applications.

Tsafrir et al. [30] have shown that user estimates are generally incorrect and estimation approaches provide significant improvements. Lee et al. [31] conducted a survey on the inaccuracy of user estimates and showed that even users with high confidence level were 35%-40% inaccurate. Our system considers the users' requests for resources in the Little Cluster, but then uses the estimates generated by the profiling system to execute them in the Big Cluster.

Rodrigues et al. [32] presented the problem of estimation/prediction and investigated the effects of machine learning algorithms for estimation in HPC systems. Further, Bharve et. al [33][34] and Bhattacharjee et al.[35] have shown that performance analysis and measurement is important in studying resource management schemes in cloud schedulers. Similarly, Palden Lama et al. [36] developed a resource provisioning model for Hadoop cluster, named *AROMA*, that uses regression to categorize an application using previously recorded performance profile. We discuss in Section VII that such an approach is complementary to RUMIG and can be integrated in cases when historical performance profiles of applications are available.

IX. Future Work

We will explore the use of Machine Learning techniques to classify users, and also estimate the resource usage for applications based on the attributes submitted for the job. We will also explore multiple migrations by a container in Mesos, once Docker is updated to allow continuous monitoring of migrated containers.

X. Conclusion

Our work, RUMIG, is the first one to present and successfully implement a design to migrate containers between nodes of a Mesos cluster for the purpose of resource right-sizing. This feature makes it possible for applications to effectively use the existing and emerging academic cloud infrastructures. RUMIG shows considerable improvements over the default setup of Aurora-Mesos. Overall, the approach can provide a gain of 65% in runtime (Arcus cluster), 51% in CPU utilization (Chameleon cloud), and 27% in memory utilization (Jetstream cloud). We quantified the overhead of migrating containers within a Mesos cluster. Apart from the scientific applications that generate a huge amount of data, wherein the data transfer costs are dominant and unavoidable, our migration scheme in RUMIG improves the overall utilization and performance. The elastic design of RUMIG provides significant improvements over the static, no-migration approach of RUMIG-base. The runtime with RUMIG decreased by 28% (Chameleon Cloud), the CPU utilization improved by 29% (Jetstream) and Memory utilization improved by 8% (Chameleon Cloud). Workloads often have repetitive behavior and do not need to be profiled every time a job is submitted. RUMIG can be configured to

reuse profiling for previously estimated tasks. As the size of the Little and Big clusters is elastic, RUMIG can dynamically turn-off the estimation step, in case it is not required or feasible for small allocations.

ACKNOWLEDGMENT

This research is partially supported by the National Science Foundation under Grant No. OAC-1740263.

REFERENCES

[1] C. A. Stewart, D. C. Stanzione, J. Taylor, E. Skidmore, D. Y. Hancock, M. Vaughn, J. Fischer, T. Cockerill, L. Liming, N. Merchant, T. Miller, and J. M. Lowe, "Jetstream," in *Proceedings of the XSEDE16 on Diversity, Big Data, and Science at Scale - XSEDE16*. ACM Press, 2016, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2949550.2949639

[2] J. Mambretti, J. Chen, and F. Yeh, "Next Generation Clouds, the Chameleon Cloud Testbed, and Software Defined Networking (SDN)," in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*. IEEE, 2015, pp. 73–79. [Online]. Available: http://ieeexplore.ieee.org/document/7421896/

[3] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *Proceedings of the 2009 conference on Hot topics in cloud computing*. USENIX Association, 2009, p. 22.

[4] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management." [Online]. Available: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2541940.2541941

[5] V. K. Vavilapalli, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, and H. Shah, "Apache Hadoop YARN," in *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13*. ACM Press, 2013, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2523616.2523633

[6] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega," in *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. ACM Press, 2013, p. 351. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2465351.2465386

[7] "Apache Aurora." [Online]. Available: http://aurora.apache.org/

[8] "Marathon: A container orchestration platform for Mesos and DC/OS." [Online]. Available: https://mesosphere.github.io/marathon/

[9] "TORQUE Resource Manager." [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/

[10] "Slurm Workload Manager." [Online]. Available: https://slurm.schedmd.com/

[11] L. T. Yang, X. Ma, and F. Mueller, "Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution," Tech. Rep., 2005. [Online]. Available: https://ieeexplore.ieee.org/document/1559992

[12] A. Wong, D. Rexachs, and E. Luque, "Parallel Application Signature for Performance Analysis and Prediction," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 2009–2019, 7 2015. [Online]. Available: http://ieeexplore.ieee.org/document/6827943/

[13] S. Sodhi and J. Subhlok, "Performance Prediction with Skeletons," Tech. Rep., 2005. [Online]. Available: http://www.cs.uh.edu

[14] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms *," 2017. [Online]. Available: https://doi.org/10.1145/3132747.3132772 https://doi.org/10.1145/3132747.

[15] Y. Cheng, Z. Chai, and A. Anwar, "Characterizing Co-located Datacenter Workloads: An Alibaba Case Study," 8 2018. [Online]. Available: http://arxiv.org/abs/1808.02919

[16] James Turnbull, *The Docker Book: Containerization is the new virtualization*, 2014. [Online]. Available: https://www.dockerbook.com/

[17] Ken McDonell, "Performance Co-Pilot." [Online]. Available: http://pcp.io/

[18] "APACHE MESOS 2016 SURVEY REPORT HIGHLIGHTS," Tech. Rep. [Online]. Available: https://mesosphere.com/wp-content/uploads/2016/11/apache-mesos-survey-2016-infographic.pdf

[19] J. G. Amar, "The Monte Carlo method in science and engineering," *Computing in Science & Engineering*, vol. 8, no. 2, pp. 9–19, 2006. [Online]. Available: http://ieeexplore.ieee.org/document/1599369/

[20] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors." [Online]. Available: http://parsec.cs.princeton.edu/publications/bienia09parsec-2.0.pdf

[21] ——, "The PARSEC Benchmark Suite Tutorial -PARSEC 2.0." [Online]. Available: http://parsec.cs.princeton.edu/download/tutorial/2.0/parsec-2.0-tutorial.pdf

[22] ——, *BENCHMARKING MODERN MULTIPROCESSORS*. Princeton University New York, 2011. [Online]. Available: http://parsec.cs.princeton.edu/publications/bienia11benchmarking.pdf

[23] "Kubernetes Vertical Pod Autoscaler." [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

[24] G. Rattihalli, P. Saha, M. Govindaraju, and D. Tiwari, "Two stage cluster for resource optimization with Apache Mesos," in *MTAGS17: 10th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*, Denver, USA, 2017.

[25] G. Rattihalli, "Exploring Potential for Resource Request Right-Sizing via Estimation and Container Migration in Apache Mesos," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 12 2018, pp. 59–64. [Online]. Available: https://ieeexplore.ieee.org/document/8605758/

[26] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, "Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers." Springer, Berlin, Heidelberg, 2015, pp. 316–323.

[27] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in Cloud Computing: State of the Art and Research Challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/7937885/

[28] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. New York, New York, USA: ACM Press, 2016, pp. 217–228. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2950290.2950328

[29] J. Hadley, Y. Elkhatib, G. Blair, and U. Roedig, "MultiBox: Lightweight Containers for Vendor-Independent Multi-cloud Deployments." Springer, Cham, 2015, pp. 79–90.

[30] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling Using System-Generated Predictions Rather than User Runtime Estimates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4180346/

[31] C. Bailey Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are User Runtime Estimates Inherently Inaccurate?" Springer, Berlin, Heidelberg, 2005, pp. 253–263.

[32] E. R. Rodrigues, R. L. F. Cunha, M. A. S. Netto, and M. Spriggs, "Helping HPC users specify job memory requirements via machine learning," *Proceedings of the Third International Workshop on HPC User Support Tools*, pp. 6–13, 2016. [Online]. Available: https://dl.acm.org/citation.cfm?id=3018836

[33] Y. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, and A. Gokhale, "FECBench: An Extensible Framework for Pinpointing Sources of Performance Interference in the Cloud-Edge Resource Spectrum," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 10 2018, pp. 331–333. [Online]. Available: https://ieeexplore.ieee.org/document/8567679/

[34] Y. Barve, S. Shekhar, S. Khare, A. Bhattacharjee, and A. Gokhale, "UPSARA: A Model-Driven Approach for Performance Analysis of Cloud-Hosted Applications," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 12 2018, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/document/8603147/

[35] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, "A Model-Driven Approach to Automate the Deployment and Management of Cloud Services," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 12 2018, pp. 109–114. [Online]. Available: https://ieeexplore.ieee.org/document/8605766/

[36] P. Lama and X. Zhou, "AROMA," in *Proceedings of the 9th international conference on Autonomic computing - ICAC '12*. ACM Press, 2012, p. 63. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2371536.2371547