2019 IEEE 12th International Conference on Cloud Computing (CLOUD)

# Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes

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

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

***Abstract*—Cloud platforms typically require users to provide resource requirements for applications so that resource managers can schedule containers with adequate allocations. However, the requirements for container resources often depend on numerous factors such as application input parameters, optimization flags, input files, and attributes that are specified for each run. So, it is complex for users to estimate the resource requirements for a given container accurately, leading to resource over-estimation that negatively affects overall utilization. We have designed a *Resource Utilization Based Autoscaling System* (RUBAS) that can dynamically adjust the allocation of containers running in a Kubernetes cluster. RUBAS improves upon the Kubernetes Vertical Pod Autoscaler (VPA) system non-disruptively by incorporating container migration. Our experiments use multiple scientific benchmarks. We analyze the allocation pattern of RUBAS with Kubernetes VPA. We compare the performance of container migration for in-place and remote node migration and we evaluate the overhead in RUBAS. Our results show that compared to Kubernetes VPA, RUBAS improves the CPU and memory utilization of the cluster by 10% and reduces the runtime by 15% with an overhead for each application ranging from 5% to 20%.**

## I. Introduction

The Cloud Native Computing Foundation (CNCF) has driven the adoption of containers in Cloud infrastructures and it is currently used by over 50,000 projects [1]. The key platform in CNCF is Kubernetes, which is also CNCF's first project. The Kubernetes project is highly active with over 2,300 unique contributors and has emerged as the most popular container orchestration platform today. Kubernetes is reportedly used by companies with massive scale computing infrastructures such as Google, Yahoo, Intel, Comcast, IBM, and eBay. Kubernetes provides important features for computing on the cloud such as Service discovery, load balancing, automatic bin packing of containers based on resource requirements, horizontal scaling of containers out of the box, and self healing – when a node fails, Kubernetes automatically re-schedules the applications on a healthy node.

However, just like other application deployment systems, Kubernetes too suffers from a resource estimation problem. In clusters and clouds, a user's estimate of required resources (e.g., CPU, Memory, GPU, I/O, Network bandwidth) and configuration of each node, or VM, for the experiments is critical in allocating VMs or bare metal nodes, and scheduling jobs. Accurate resource estimation is a challenge for end users as the requirement for applications can vary in each run because it is dependent on various configuration values such as the input file sizes, optimization flags, input parameter choices, and the core application kernel.

Errors in resource requirements for applications, specified by users, is a well known problem and is not restricted to the use of Kubernetes [2] [3]. Cloud resource managers such as YARN [4], Omega [5], Mesos schedulers such as Apache Aurora [6] and Marathon [7], along with HPC resource managers such as Torque [8] – all require estimation for each application before it is launched.

It has been recorded that users in about 70% of cases request more resources than required [9]. In [9], Delimitrou et al. also argue that over-allocation of resources for applications causes increased wait times for pending tasks in the queue, reduced throughput, and under-utilization of the cluster. Incorrect estimation of resources can significantly increase the overall cost, in Service Units (SUs), charged by cloud platforms running a set of applications [10].

We also analyzed the Azure Public Dataset [11][12], that was released by Microsoft. In Figure 1 (top subplot), we show the average CPU utilization and peak average CPU utilization over a 24 hour period. The average CPU utilization is always under 10%, while the peak utilization hovers between 14-17%. In Figure 1 (bottom sub-plot), for data collected over 30 days, we show the number of Virtual Machines that had average CPU utilization in 25% ranges. We again see that over 1.5 million Virtual Machines had average CPU utilization between 0-25%.

Kubernetes attempts to address the resource estimation problem via the Vertical Pod Autoscaler (VPA) [13] feature, which does not require any user input on resource requirement. The VPA generates an estimate based on the resources the application is currently using and then corrects it by re-scheduling the application with the newly estimated resources.

2159-6190/19/$31.00 ©2019 IEEE
DOI 10.1109/CLOUD.2019.00018

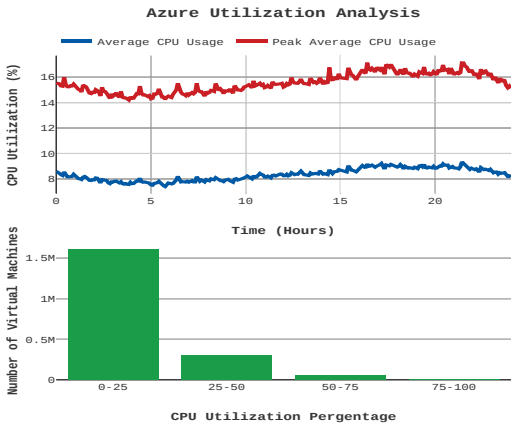


Fig. 1: *Azure Cluster CPU utilization analysis: the top graph shows the cluster peak average and average CPU utilization over 24 hour period. The bottom figure shows the number of virtual machines with average CPU utilization in the respective ranges over 30-day period.*

However, it has a *disruptive* approach. Its correction approach requires killing the application and then re-scheduling it with estimated resources. While this approach works for stateless services, it is a serious drawback for stateful and performance sensitive applications.

*Therefore, this paper explores new ways to augment container orchestration in Kubernetes that dynamically estimates the amount of resources an application requires, non-disruptively migrates, updates the resource allocation, and continues with the application execution.*

We have designed a flexible and configurable framework, Resource Utilization Based Autoscaling System (RUBAS), to address the resource correction problem in a Kubernetes setup. RUBAS starts with the users' resource requests for each application and launches it on the cluster. It then monitors the application's resource usage via Metrics Server at a configurable interval with the default set to 60 seconds. If an application's usage is beyond 10% of the allocated resources, then RUBAS creates a new estimate, checkpoints the container in which the application is running, and re-schedules it on the cluster with the new estimate. Each new estimate is calculated as the sum of median and a *buffer*, where buffer is defined as the positive deviation of the observation. This approach gives us a better estimate compared to the Kubernetes VPA, which uses the peak values rather than central tendencies. The peak observation could be an outlier in the observed data and could result in over-allocation. The sum of median and positive deviation also negates the effects of outliers. RUBAS can re-schedule on the same or a different node in the cluster. The monitoring interval is a configurable parameter and can be set by the cluster administrator. RUBAS uses *Checkpoint Restore in Userspace (CRIU)* to create checkpoints and restore docker container execution, and therefore, does not require the application to be killed and restarted. RUBAS migrates workloads to nodes/cluster that have the required resources. Unlike Kubernetes, it does not assume that the resource allocation in an existing node can be scaled up seamlessly. We note that the VPA approach of vertical auto scaling is not possible with cloud resource scheduling tools such as Mesos because allocation changes are not possible once an application has been scheduled. However, the RUBAS approach could be extended to Mesos.

While RUBAS works best for target applications that do not have frequent resource variations, its design can also cater to those with frequent variations via multiple migrations. We note that several HPC applications have a resource consumption pattern wherein the resource usage does not vary frequently after the initial phase [14].

Specifically, we make the following contributions:

- *This paper describes design and implementation of a new non-disruptive and dynamic vertical scaling system for Kubernetes, called RUBAS.*
- *We study and quantify the design trade-offs in different design choices on the performance of RUBAS including profiling interval, container migrations and application restart overheads.*
- *We present results and analysis of RUBAS performance compared to the Kubernetes VPA in terms of CPU and memory allocation patterns, application runtime, and overall cluster resource utilization.*

## II. Design

### A. Kubernetes Architecture

In Figure 2, we show the components of a typical Kubernetes setup. It has a Master and Worker nodes along with *etcd* for maintaining state of the Kubernetes cluster. The master node consists of an *API server*, *Scheduler*, and a *Controller Manager*. The controller manager is responsible for the core functions of Kubernetes. These include replication controller, endpoints controller, namespace controller, and service accounts controller. The API server helps in validation and configuring the data for API objects such as pods, and services. The scheduler has many policies and is topology aware. It uses the topology information to appropriately schedule the pod on a node based on its policy requirement. The worker nodes in Kubernetes consist of a *kubelet*, proxy system, and a Docker runtime environment. The kubelet is responsible for communications between the worker node and the master node. It maintains the state of the worker node and executes the pods as defined in the *PodSpec* provided by the master. The PodSpec is a YAML or JSON object that describes the pod. Kubelet reads the PodSpec and executes the pod and its corresponding container via Docker. The proxy is used to communicate between the pods and sets up the network infrastructure [15].

### B. RUBAS design

In Figure 3, we list the steps involved in executing a job. An application is first submitted to RUBAS. RUBAS creates the launch specification required for the job to execute on the Kubernetes cluster, which includes an unique-id to monitor the application's resource usage. The specification

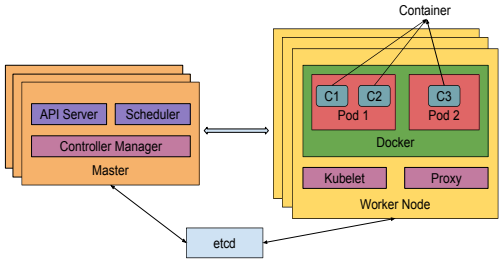


Fig. 2: *General architecture of Kubernetes showing different components of a typical Kubernetes setup.*

is sent to Kubectl, which is a Kubernetes component for managing the cluster. Kubectl forwards the job to Kubernetes Master. Kubernetes Master schedules the application on the worker nodes. In step 5, the metrics server collects resource utilization details of each container running on the cluster. In step 6, RUBAS checks the utilization of each container against the allocation. If the allocation and the utilization matches, then RUBAS allows the execution to continue. If the allocation and utilization does not match, then in Step 7, RUBAS sends two instructions to Docker – one to checkpoint the container and another to create an image that contains the data generated by the container. Docker then creates these images in step 8. In step 9, the image and checkpoint are stored in NFS, so that they are accessible across the cluster. In step 9a, the information of the created checkpoint and the image is sent to RUBAS. In step 10, RUBAS creates the launch specification with the checkpoint and the image and sends it to kubectl. Kubectl forwards the job to Kubernetes master, which then schedules the job on the worker nodes. In step 13, the checkpoint and the image is downloaded to the selected machine and the container is restored.

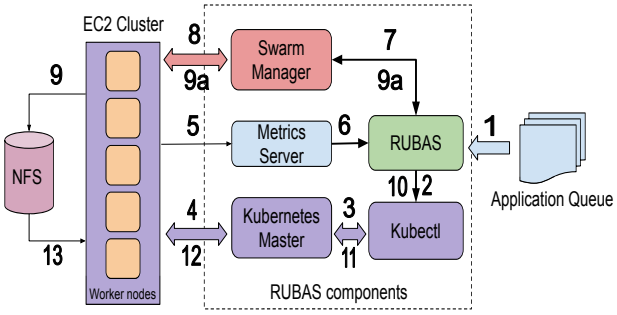


Fig. 3: *Architecture of RUBAS: workflow of profiling, checkpointing and resuming applications in the cluster.*

## III. Vertical Auto-scaling

### A. Discussion- Kubernetes VPA

The current implementation of VPA does not require the user to provide any resource requests, for CPU and memory, for a pod. By default, the initial allocation in Kubernetes is a configurable lower limit or the minimum guaranteed resources allocated for a pod. The VPA uses a recommender, which tracks the historical usage of the pod and suggests a new limit whenever needed. The default formula used by the recommender for new resources is the following:

$$\textit{New Resources} = max\{\textit{peak + MinBumpUp, peak x BumpUpRatio}\}$$

where

$$\textit{peak = max memory used in observed data,}$$

$$\textit{MinBumpUp = 100MB \& BumpUpRatio = 1.2}$$

While these equations are modifiable, this resource correction approach itself has drawbacks. For example, the peak memory usage could have been a temporary spike in the application, but VPA aims to allocate for that amount throughout the application's duration. Furthermore, the current VPA implementation restarts the pod every time the allocation is changed. Such a disruptive process results in a loss of the state of the containers. Though Docker released a *Docker update*, to allow updatation of the resources allocated to the containers, it cannot be directly implemented with large distributed systems like Kubernetes. This is because there are various components such as scheduler, control manager and kubelet that depend on the original allocation to take decisions. Unless all these components are aligned, it is currently difficult to leverage the new Docker update feature for VPA.

### B. Rresource Estimation: RUBAS vs. VPA

- RUBAS has two key modules:
  - A runtime profiling and estimation system. In earlier work, we have shown the effectiveness of a similar runtime profiling system [16] [17].
  - A vertical scaling system that can change the allocation at runtime non-disruptively (if required).
- RUBAS schedules all the applications as they arrive on the cluster and starts collecting resource usage data at one second intervals.
- The profiling data collection involves CPU and memory usage and is obtained via the Metrics Server. Based on this data, a new estimation is generated every 60 seconds.
- Based on the data collected every 60 seconds, an estimate is generated using the following formula:

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

$$\textit{Required Resource} = \textit{Median Of Observations} + \textit{buffer}$$

  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 the absolute deviation of the observed values.
- As the applications are initially scheduled without any resource allocation, we set the limits via these patching options:
  - Disruptive: The applications are restarted upon patching. This is used when an application restart does not affect the outcome.

– Non-Disruptive: The applications are checkpointed and restarted using the checkpoint. This process may involve container migration. If the estimated amount of resources are not available on the same node, the application is migrated to another node that has the availability. However, in case there are no nodes with the available resources, the application continues execution in its current state.

- The process of estimation and patching continues until the application terminates.

## IV. Evaluation

### A. Evaluation Setup

*1) Setup:* The cluster setup for the experiments consisted of 30 Kubernetes worker nodes and 1 Kubernetes master node, all in the AWS cloud. The details of the setup are presented in Table I. Table II provides details about the benchmarks used as workloads in the experiments.

| Equipment/OS/Software | Description/Version |
|---|---|
| AWS EC2 Nodes (31) - | Compute Optimized c5.2xlarge, 8 core processor at 2.3GHz, 16 GB DDR4 RAM |
| Operating System | Ubuntu 18.04.1 LTS |
| Docker | 17.06.1 |
| Kubernetes | 1.12.3 |

TABLE I: *Description of the infrastructure used in the experiments on Amazon AWS EC2 Cloud cluster.*

| 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 |

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

### B. Determining the best monitoring interval

The monitoring interval has a significant effect on the resource estimation and needs to be determined experimentally for each class of applications. We conducted experiments for monitoring intervals ranging from 15 seconds to 90 seconds. Figure 4 presents the results at intervals 15, 30 and 60 seconds along with the runtime. In these experiments we initially allocated resources based on static profiling. The number in each bar represents the number of container migrations that were performed before the completion of execution. We observe that at 15 seconds, there are several container migrations that results in high overhead. For example, DGEMM has 8 container migrations and the runtime is almost 5 times the runtime for DGEMM when it is run with optimal resources as determined by static profiling. The overhead can be attributed to estimation error due to insufficient data points. At the 30 second interval, we see a reduction in the number of container migrations and the runtime improves. However, it is still higher than the runtime achieved when the optimal resources are allocated. At the 60 second interval, we observe the least number of container migrations and the runtimes are very close to the runtimes obtained with optimal resource allocations. The overhead due to the estimation and container migration on the benchmarks ranges from 5% to 20%. At 75 and 90 second intervals, we do not see any improvement for either the number of container migrations or the runtime. We thus conclude that 60 seconds is ideal monitoring interval for the given set of benchmarks.

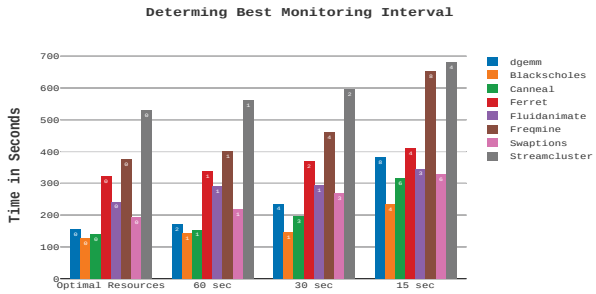


Fig. 4: *Determining the best monitoring interval. The number in each bar represents the migrations for allocations corresponding to different monitoring intervals.*

### C. Comparison with Kubernetes VPA

In Figure 5, we provide a comparison with Kubernetes VPA for runtime and number of corrections. The presented results are average of 10 executions. We conducted experiments for 3 different settings: (1) RUBAS without resource allocation restrictions; (2) Kubernetes VPA without resource allocation restrictions; and (3) 30% resources beyond the optimal requirement, as determined by static profiling Kubernetes VPA. The results show that even with grossly incorrect allocations, RUBAS is able to estimate and correct the allocation in all the applications. In comparison, Kubernetes VPA performs multiple restarts, which increases the overhead and in some cases the runtime is more than twice that of RUBAS. However, when the resource allocation is closer to the optimal resources, Kubernetes VPA and RUBAS have similar performance. In the case of Blackscholes and Canneal, Kubernetes VPA peforms better than RUBAS. It is to be noted that for Kubernetes to outperform RUBAS, the user needs to know in advance the near optimal resource requirement for each application. When such information is not available, RUBAS is a better choice compared to Kubernetes VPA.

### D. Comparison of allocation, RUBAS and Kubernetes VPA

In Figure 6, we show the allocations made by RUBAS and Kubernetes VPA at various stages of execution for the DGEMM application. Kubernetes VPA takes 4 corrections for the DGEMM benchmark to reach the optimal CPU allocation whereas RUBAS takes only 2 corrections. For memory allocations, Kubernetes VPA takes 3 corrections while RUBAS

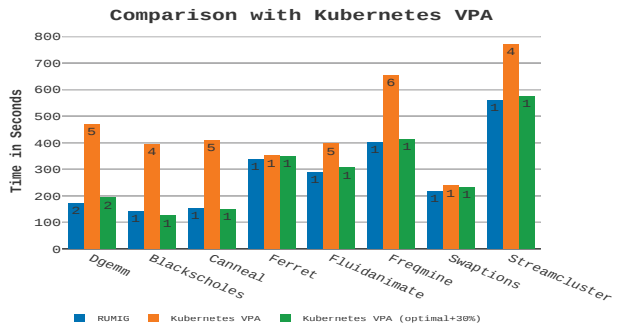


Fig. 5: *RUBAS vs. Kubernetes VPA: runtime and number of corrections.*

takes 2 corrections. As RUBAS has more accurate resource estimates it performs fewer corrections. Due to the inclusion of container migration, the runtime is lower by 63% in the case of RUBAS. Kubernetes VPA restarts after every correction, whereas RUBAS resumes from the checkpoint.

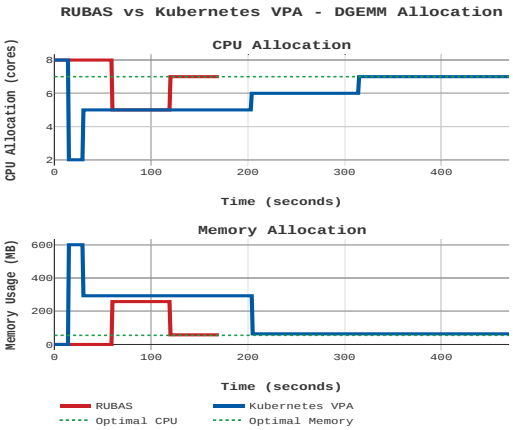


Fig. 6: *CPU and memory allocation with estimation for DGEMM benchmark in RUBAS and Kubernetes VPA setups.*

### E. Disruptive vs Non-Disruptive Vertical Scaling

In Figure 7, we show the difference in runtime while using the RUBAS estimation methodology. For the experiment we used 60 second monitoring interval as explained in Section IV-B. In all cases, we see an improvement in runtime when using migrations. For DGEMM, we see a 49.10% decrease in runtime, Blackscholes has 23.65% decrease, Canneal has 23.5%, Ferret has 11.97%, Fluidanimate has 3.33%, Freqmine has 7.81%, Swaptions has 13.49%, and Streamcluster has 4.91% decrease in runtime. As the size of the of the migration image increases, the runtime gain decreases as can be seen with applications such as Fluidanimate, Freqmine and Streamcluster where the gains are less than 10%. On the other hand, the benefits of migration are more pronounced when the initial estimations are either incorrect or the application changes its requirement. For example, in DGEMM, the application slowly increases its usage and this leads to incorrect estimations, which then causes multiple migrations/restarts.

In Figure 8, we compare the migration costs of RUBAS for in-place scaling on the same node versus scaling on another node. We used the *nodeSelector* option in the Kubernetes PodSpec to control in-place vs remote node migration. By pinning the pod to a particular node we were able to perform in place migrations, which resulted in runtime gains. We notice a 4.1% decrease in runtime for DGEMM, 2.09% for Blackscholes, 2.61% for Canneal, 5.62% for Ferret, 6.89% for Fluidanimate, 5.73% for Freqmine, 4.58% for Swaptions, and 2.13% for Streamcluster.

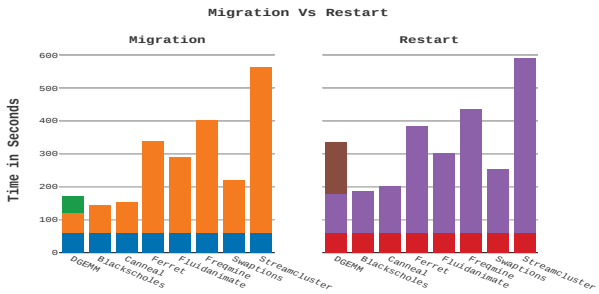


Fig. 7: *Migration vs Restart with interval size of 60 seconds.*

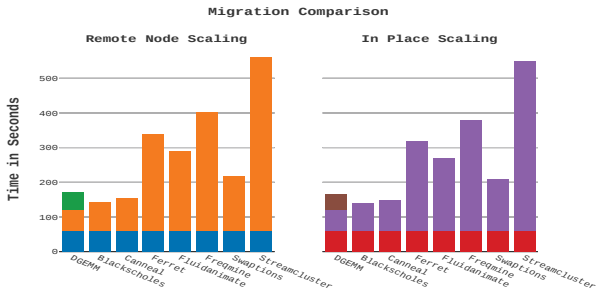


Fig. 8: *In-place scaling vs remote node scaling.*

### F. Cluster utilization

*1) Setup:* We ran a total of 10 iterations, where each iteration consisted of 135 instances of each benchmark presented in Table II, i.e. a total of 1080 jobs. The experiment was executed on a cluster of 30 *c5.2xlarge* nodes as described in Table I. Apart from the execution process explained in the section II-B, the cluster experiments had some additional conditions for migration. If the estimated resource requirement was more than the available resources in the cluster, then the application was allowed to continue execution, as it is better to continue execution rather than wait in the queue till the cluster has available resources. In the meanwhile, when the estimated resources was generated, and if it was determined that the application needs to be scaled, then a checkpoint was created regardless of the resource availability. The idea was to address situations when an application fails due to incorrect resource allocation while it is allowed to continue execution.

*2) Performance (runtime):* In Figure 9, we show the average runtime of default Kubernetes, Kubernetes VPA and RUBAS. All three setups were provided with the same set of applications with the same resource requests, i.e. optimal + 30% resources. In our experiments, we see that default Kubernetes, with no ability to modify the resources, takes

much longer to complete. On average, default Kubernetes took 32162 seconds. In comparison, Kubernetes VPA has a much better runtime of 23127 seconds on average. This accounts to about 28% decrease in runtime. Kubernetes with VPA performed much better than the default Kubernetes even by restarting the pods. Restarting the pods should have been a detrimental step but because VPA is able to correct the allocation and make resources available, more applications are launched by Kubernetes resulting in lower runtime. Our approach, RUBAS, achieves improvement by avoiding the overhead of restarts by incorporating container migration. We see that with RUBAS the runtime further reduces to 19457 seconds. This accounts to about 16% improvement over Kubernetes VPA and about 40% improvement over default Kubernetes.


Fig. 9: *Runtime results for a cluster with 30 nodes, running 1080 jobs to completion.*

*3) CPU utilization:* In Figure 10, we show the CPU utilization corresponding to the allocation. In default Kubernetes, we see that only about 47% of the allocated CPU was used by the applications. This means that more than half of the allocated CPU was idle. As there are several application waiting in the queue, the under utilization of resources effectively increases the overall runtime. Kubernetes with VPA was able to correct the allocation and thus had a much better utilization at about 72%. However, Kubernetes VPA has a drawback in the way the applications are restarted. Instead of restarting the application immediately, we observed that Kubernetes puts the task in the queue, which delays the execution of the application. In situations where the application is providing a service, there could be a long gap till the application is restarted. We address this issue in RUBAS. The application that needs to be resumed has the highest priority in the RUBAS queue. With RUBAS, using container migration, we see even better CPU utilization at about 82%.

*4) Memory utilization:* In Figure 11, we show the average memory utilization of the cluster with default Kubernetes, Kubernetes with VPA, and RUBAS. The trends are similar to the CPU utilization data presented in Section IV-F3. Default Kubernetes had an average memory utilization of 43% when the allocation was 100%. Kubernetes VPA improved the utilization to 76% as it was able to re-size the applications. RUBAS further improved the utilization to 86%.

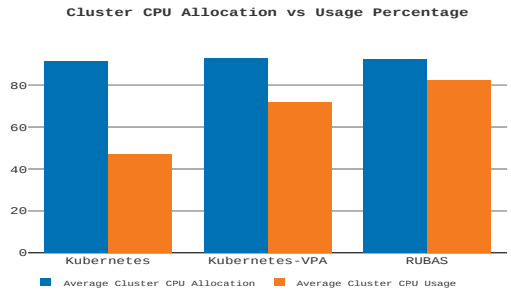


Fig. 10: *CPU utilization results for a cluster with 30 nodes, running 1080 jobs to completion.*

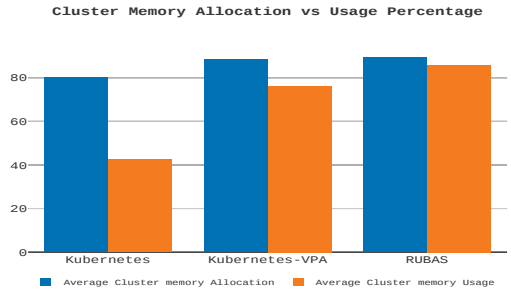


Fig. 11: *Memory utilization results for a cluster with 30 nodes, running 1080 jobs to completion.*

### G. Effects of Migration- Discussion

*1) Ability to vertically scale for ad-hoc jobs:* Vertical scaling of service jobs does not require migration, as the services generally do not need to maintain the state after restart. Also, service jobs generally have replicated jobs that help with load balancing. So a restart for these jobs is not very critical. However, ad-hoc or one-time jobs need to maintain the state of execution or else they will have to start execution again from the beginning. We have shown that RUBAS can migrate ad-hoc jobs and continue the execution while being able to vertically scale them.

*2) Ripple Effect of Migrations:* Ripple effect happens when the container keeps migrating from one machine to another without any actual work being done. This happens when the time frame for estimation is small and the estimation is inaccurate. We see such an example for the 15 second interval in Figure 12, which has 8 migrations and the runtime is twice the runtime of 60 second interval.

*3) Migration is Not Always the Best Choice:* There are some situations where container migration is not the best choice. For state-less applications, it not required to maintain state. So, instead of migration, restart of the application with the appropriate resources is a better choice. Most state-less applications are often replicated for the purpose of load balancing. If one of the replicated applications is restarted, there will be a momentary degradation of the service until a new replica takes over.

*4) Stability with Multiple Container Migrations:* In our experiments we observed that when an application is migrated more than 4 times, then the migration image is sometimes generated incorrectly. This is a current limitation of CRIU.

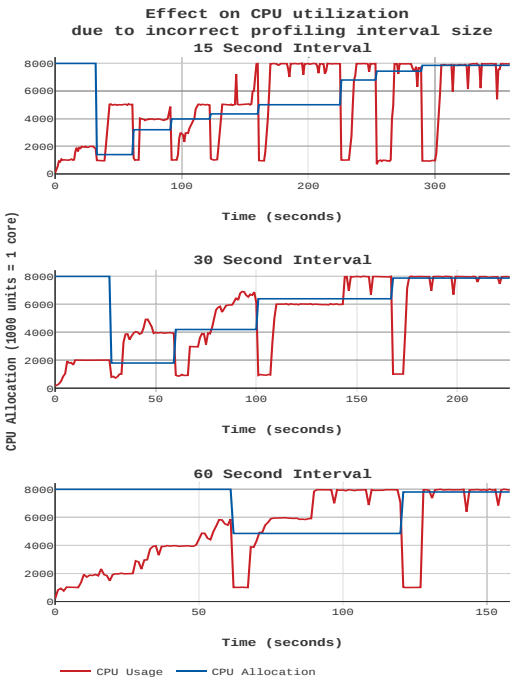


Fig. 12: *Effects of incorrect estimations caused by smaller monitoring interval. Each drop in CPU utilization is due to migration (8 in total for 15 second interval). The benchmark used is DGEMM and monitoring interval is 15, 30, and 60 seconds.*

*5) Incorrect estimations could lead to higher execution time:* In applications that generate a lot of data, the migrations times are high as the generated data needs to packaged in a Docker image and then copied to the target machine. This effect is quantified in Figure 8, where we observe considerable increase in runtime for Ferret, Fluidanimate and Freqmine applications due to migration. In Figure 12, at 15 second interval we have a runtime of over 300 seconds for DGEMM, while at 60 second interval it is less than half. The smaller monitoring interval is causing to incorrect estimations and that leads to higher runtime.

*6) Migration Downtime:* Container migration involves multiple steps, which are time consuming. These steps include:

- Checkpoint Creation: This process is taken care of by CRIU. It creates a checkpoint of the current state of the Docker container. However, it does not checkpoint any changes that the Docker image has made on the storage.
- Creating checkpointed Docker image: As CRIU does not create a checkpoint for the data generated by the container, RUBAS needs to handle it separately. RUBAS creates a new Docker image from the running container and copies all the data that is generated by the container. This process is time consuming and the overhead depends on the generated size of the data.
- Transferring checkpoint and checkpointed Docker image to the target machine: Once the application is re-scheduled to be launched on a node by Kubernetes, we need to transfer the checkpoint and the checkpointed Docker image on the target node. This could be expensive if the application is relaunched on a different node. We note this behaviour in Figure 8 for Ferret, Fluidanimate, and Freqmine where the transfer time is about 15-20 seconds. This is due to the size of the checkpointed Docker image and the network bandwidth between the two nodes.
- Resuming from the checkpoint: CRIU enables resumption of the container on the target node. CRIU uses the newly created Docker image from the checkpoint instead of the original Docker image.

| Benchmarks | Size (MB) | Chk Creation | Image creation | Transfer | Resume |
|---|---|---|---|---|---|
| 1. Blackscholes | 74.5 | <1 | 1 | 1 | <1 |
| 2. Canneal | 63.36 | 1 | 1 | 1 | <1 |
| 3. Ferret | 636.6 | 2 | 13 | 16 | <1 |
| 4. Fluidanimate | 733.26 | 2 | 12 | 17 | <1 |
| 5. Freqmine | 820.14 | 3 | 13 | 20 | <1 |
| 6. Swaptions | 236.3 | 2 | 6 | 7 | <1 |
| 7. Streamcluster | 412.6 | 2 | 7 | 8 | <1 |
| 8. DGEMM | 101.2 | 1 | 3 | 1 | <1 |

TABLE III: *Average downtime of each benchmark at different stages (Unit: second).*

## V. Related Work

The area of container migration is still in its infancy, but virtual machine migration is a well studied topic. There are many projects on scheduling and VM migration in the cloud with focus on the effect bin packing algorithms cloud [18], energy aware scheduling, and migration prediction for virtual machine migration in the cloud [19].

The pMapper [20] project studied continuous dynamic allocation and optimization on virtual machines(VM). They have a similar approach as RUBAS, but in the context of virtual machines.

There are many projects that purely concentrate on vertical scaling in the context of VMs on various cloud platforms and explore different aspects of vertical scaling such as application driven vertical scaling, vertical scaling across federated clouds, and on specific platforms such as OpenStack [21], [22]. Han et. al. [23] presented a lightweight resource allocation system that uses the resource utilization of applications running in the VM to inform scheduling decision for VM allocation.

Other projects such as [24], [25] show the effects of using applications usage to aid VM scheduling decisions. The positive effects of dynamic vertical and horizontal scaling of virtual machines and containers has also been studied [26]. Our work involves enabling vertical scaling for stateful applications i.e. vertical scaling for containers where continuation of execution is important. DoCloud [27], presents a proactive method of performing vertical scaling of Docker containers that run web applications. They pro-actively vary the resource allocation for containers to attain higher utilization. Hadley et. al. [28] have shown the capability of CRIU (Checkpoint Restore In Userspace) across multiple clouds. They show this capability for both stateful and stateless applications. However, we are not aware of any work that combines container migration capability for stateful applications along with vertical

scaling. This paper demonstrates this new capability and explores its efficacy when used in a Kubernetes environment.

## VI. Conclusion

RUBAS provides an effective correction for resource allocation for applications running inside a Kubernetes cluster. We compare RUBAS with the current state of art, Kubernetes Vertical Pod Autoscaler (VPA), which also aims at providing accurate resource allocation. RUBAS provides many benefits over Kubernetes VPA by providing a non-disruptive method of auto-scaling, a more accurate estimation methodology which together improve the cluster CPU and memory utilization by 10%, and reduced runtime by 15%. We see an overhead for each application runtime in the range of 5% to 20%, but the applications have near accurate allocations which allows the cluster to be utilized by other applications. RUBAS also decreases the number of restarts over Kubernetes VPA. We also, quantify the overhead of container migration in RUBAS at various stages of migration, conduct experiments to determine the best interval size for monitoring the container for the purpose of estimations, compare the allocation pattern of RUBAS with Kubernetes VPA, and evaluate the overhead of In-Place scaling and remote node scaling. Overall, even with grosly inaccurate allocations, RUBAS is able to correct the allocations with much less number of corrections than Kubernetes VPA while providing much improved cluster utilization.

## Acknowledgment

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

## References

[1] "Cloud Native Computing Foundation." [Online]. Available: https://www.cncf.io/

[2] W. Lin, J. Z. Wang, C. Liang, and D. Qi, "A Threshold-based Dynamic Resource Allocation Scheme for Cloud Computing," *Procedia Engineering*, vol. 23, pp. 695–703, 1 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705811054117

[3] F. Chang, J. Ren, and R. Viswanathan, "Optimal Resource Allocation in Clouds," in *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, 7 2010, pp. 418–425. [Online]. Available: http://ieeexplore.ieee.org/document/5557966/

[4] 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

[5] 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

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

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

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

[9] 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

[10] 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.

[11] 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.

[12] "Azure Public Dataset." [Online]. Available: https://github.com/Azure/AzurePublicDataset

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

[14] 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/

[15] "Reference - Kubernetes." [Online]. Available: https://kubernetes.io/docs/reference/

[16] G. Rattihalli, P. Saha, M. Govindaraju, and D. Tiwari, "Two stage cluster for resource optimization with Apache Mesos," 5 2019. [Online]. Available: http://arxiv.org/abs/1905.09166

[17] 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/

[18] Z. Zhang, L. Xiao, X. Chen, and J. Peng, "A Scheduling Method for Multiple Virtual Machines Migration in Cloud." Springer, Berlin, Heidelberg, 2013, pp. 130–142.

[19] V. Kherbache, E. Madelaine, and F. Hermenier, "Scheduling Live Migration of Virtual Machines," *IEEE Transactions on Cloud Computing*, p. 1, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8046068/

[20] A. Verma, P. Ahuja, and A. Neogi, "pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems." Springer, Berlin, Heidelberg, 2008, pp. 243–264.

[21] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni, "Application-driven dynamic vertical scaling of virtual machines in resource pools," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/6838238/

[22] M. Turowski and A. Lenk, "Vertical Scaling Capability of OpenStack." Springer, Cham, 2015, pp. 351–362. [Online]. Available: https://www.springerprofessional.de/vertical-scaling-capability-of-openstack/2520776

[23] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight Resource Scaling for Cloud Applications," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE, 2012, pp. 644–651. [Online]. Available: http://ieeexplore.ieee.org/document/6217477/

[24] D. A. Bacigalupo, J. van Hemert, X. Chen, A. Usmani, A. P. Chester, L. He, D. N. Dillenberger, G. B. Wills, L. Gilbert, and S. A. Jarvis, "Managing dynamic enterprise and urgent workloads on clouds using layered queuing and historical performance models," *Simulation Modelling Practice and Theory*, vol. 19, no. 6, pp. 1479–1495, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X11000153

[25] J. Z. Li, J. Chinneck, M. Woodside, and M. Litoiu, "Fast scalable optimization to configure service systems having cost and quality of service constraints," in *Proceedings of the 6th international conference on Autonomic computing - ICAC '09*. New York, New York, USA: ACM Press, 2009, p. 159. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1555228.1555268

[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] C. Kan, "DoCloud: An elastic cloud platform for Web applications based on Docker," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2016, p. 1. [Online]. Available: http://ieeexplore.ieee.org/document/7423439/

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