Collocating: Evaluation of Overcommitting Batch Jobs in Cluster

Mondo Jiang, Zihao Zhu

1 Introduction

In recent years, the evolution of cloud services has revolutionized the landscape of of cluster provider's infrastructure [1]. To accommodate various size of business and individuals, substantial works [14][5][4][9][10] recently ensure the flexibility and scalability of cluster provider.

In this project, we plan to analyze different schedulers' behavior when scheduling low priority workloads (e.g. machine learning training jobs, daily conclusion data analysis jobs, etc.) and latency-critic workloads (e.g. web servers, machine learning inference and real-time interactive data analysis jobs) together in single cluster.

In previous works, most schedulers only schedule single type of workloads: In Mesos, each workload has its own scheduler [5], and INFaaS [10] only considers machine learning inference workloads, etc. But we might achieve better resource utilization by having a centralized scheduler to manage various types of workloads.

Emerging frameworks [12][2][7] address this challenge by over-commitment, a successful concept in many business model. Over-commitment, in essence, involves allocating resources beyond their physical capacity, leveraging statistical multiplexing and dynamic resource management to optimize utilization. This approach draws inspiration from industries where resources are constrained or demand is highly variable, such as airlines overselling seats or hotels oversubscribing rooms. By applying similar principles to computing environments, over-commitment frameworks aim to maximize resource utilization without compromising performance or reliability.

By co-locating batch jobs and real-time data analysis, we may further increase data locality among different jobs which manipulate the same partition of data, and reduce the cost of data transfer. This is especially useful in cloud data center scenario, because it can significantly reduce the overall cost for running a data center.

We plan to use Koordinator [12] as our scheduler, and compare its performance with default K8s scheduler. We began by setting up an environment on Google Kubernetes Engine(GKE) and insert a Koordinator [12] to manage overcommitment strategies. Leveraging Kubernetes's capabiulities for container orchestration and resource management, we configured the environment to dynamically colocating distinct workloads. With the infrastructure in place, we proceeded to evaluate the effectiveness of a colocating strategy by simulating workloads by Looksbusy [3], then submitting Spark batch jobs to the Kubernetes cluster. Through our testing and analysis, we assessed the impact of utilizing such strategy in terms of resource utilization, performance, and scalability. By comparing the result against the Kubernetes' resource management, this work provide evidence that over-commitment and colocating strategies can significantly enhance resource utilization and efficiency.

2 Background

Despite the advanced optimization features of Kubernetes, a significant challenge remains in the discrepancy between resource allocation and utilization. This gap arises from the disparity between client demands, typically driven by estimate peak usage, and the long-term resource requirements of certain services. As clients lack full control over user behavior over time, the utilization of allocated resources fluctuate over the life span of the application across a period of time, this example is illustrated in Fig. 1. Use the web server as an example that demonstrates the characteristic that the requests from users may peak at noon and gradually fall in the evening. In the night time, low utilization of the requested resources contribute the majority of waste.

In the context of this project, the terms "colocation" and "over-commitment" are often used interchangeably; however, they actually refer to distinct concepts. Colocation specifically refers to scheduling different workloads within a single container. Over-commitment, on the other hand, involves assigning workloads to clusters such that the total sum of requested resources exceeds the total allocatable resources at typical.

3 Design

To evaluate the workloads, we established a Kubernetes cluster with automated creation and teardown procedures using Terraform [13], deployed on the Google Cloud Kubernetes Engine. We successfully deployed Koordinator [12] and Spark Operator on our cluster. The architecture

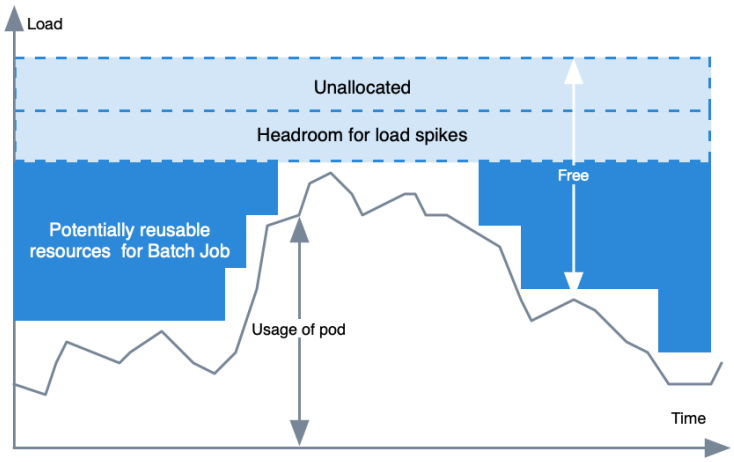

Figure 1: Illustration of the fluctuated request resource utilization over period of time

overview is depicted in Fig. 2.

3.1 Control Plane

Within the cluster, we configured Koordinator (scheduler, descheduler, controller manager) and Spark Operator. Both contain a Controller [8] component responsible for synchronizing workload configuration from Kubernetes internal object storage (etcd) to the cluster. Koorinator also contains a scheduler which handles workload scheduling (through selectors), including Spark workloads.

Spark Operator listens to the desired state for Spark applications defined in etcd, delegating the actual scheduling to the assigned scheduler. After submiting a spark application configuration through Kubernets API, Spark Operator will try to tell scheduler to set up corresponding Spark Driver and Spark Executors for this application.

We set up two namespace for running experimental group and control group. For the experimental group, all workloads are scheduled by Koordinator scheduler.

For internal authorization, Spark Operator and Koordinator have the ”editor” role for the entire cluster, simplifying testing procedures.

3.2 Schedulers

Alongside Kubernetes' default scheduler, Koordinator's scheduler operates independently as another option for workloads.

Kubernetes' default scheduler ensures every workload gets its requested resources, rejecting new workload requests when free resources are insufficient, and waiting until there is enough space for new workload. Through eviction policy we could force a new workload to be scheduled with the cost of killing other pods. In our setup, we disabled this behavior to avoid affecting existing workloads.

Koordinator's scheduler profiles real-time actual resource usage through Koordlet and aims to use the free space between requested resources and actual usage to maximize overall cluster resource usage. We set memory and CPU reclaim thresholds to 65% for testing, leaving 35% of CPU and memory requests un-colocatable for other workloads as a headroom for spikes.

3.3 Workloads

Two workload types were tested: applications and batch jobs.

Application workload was simulated using looksbusy [3] as a single pod with varying CPU and memory requests on host nodes.

For batch jobs, we used two examples provided by Spark to simulate simple/short batch jobs (using SparkPI [11]) and intense batch jobs (using SparkTC [11]).

Spark workloads were used to test Koordinator's over-commitment capabilities. In this scenario, pods occupy the cluster requesting more resources than needed. This stable stress setup demonstrates the benefits of utilizing Koordinator's over-commitment feature. With insufficient allocatable resources, overall CPU utilization remains low, providing room for Koordinator to co-locate jobs and improve resource utilization.

4 Evaluation

To assess the performance of the colocating scheduler in comparison to other schedulers, we conducted a series of

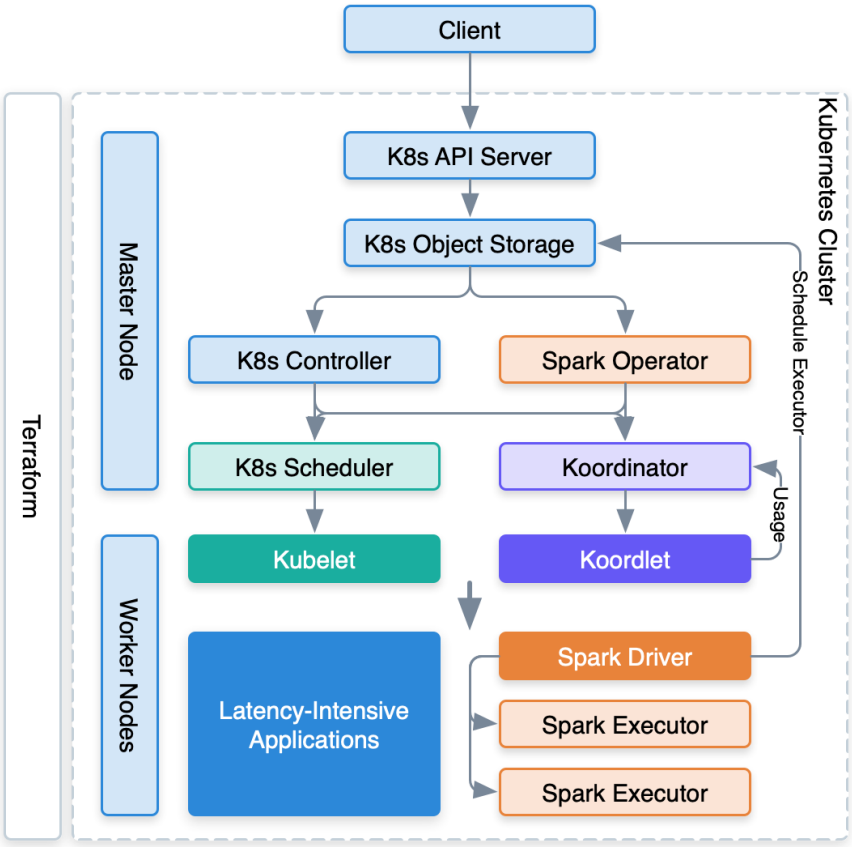

Figure 2: Architecture

experiments focusing on overall CPU and memory usage. These metrics are critical for evaluating scheduler efficiency and effectiveness under various loads and conditions.

We designed experiments to simulate a cloud cluster environment with colocated online/online and online/offline workloads across different cluster settings.

4.1 Cluster Configuration

We tested various cluster settings and made an interesting finding: Kubernetes itself (including the Host OS) requires nearly 1 CPU core and 2GB of memory for each node. Initially, we started with a 1C2G (1 core, 2GB memory) single-node cluster, but it turned out that Kubernetes couldn't even schedule its essential components!

Due to cost concerns, we settled on a 6C10G single-node cluster for our experiments. All Kubernetes components, Spark Operator, Koorinator, and workloads run on the same node but are separated by different namespaces.

Resource details are shown in Table 1.

The total allocatable CPU is 5.92 Cores, and allocatable memory is 7.26 GB.

	Capacity	Allocatable	Requested
CPU (Cores)	6	5.92	0.695
Memory (GB)	9.38	7.26	1.05

Table 1: Cluster Usage before any other workloads

4.2 Workload Settings

We set up two sets of experiments to simulate high/low usage applications followed by easy/intense Spark workloads. Details are provided in Tables 2 and 3.

4.3 Result

In both sets of experiments, the application workload requests almost all the allocatable CPU resources. Kubernetes' default scheduler consistently rejects Spark workloads (as shown in Table 4). However, when we switch to Koorinator, Spark workloads can be executed successfully, utilizing nearly all potentially reusable resources and increasing overall cluster usage.

No.	Scheduler	CPU Req.	Mem Req.	CPU Avg. Usage	Mem Avg. Usage
1.1	Default	3.7C	200MB	3C	100MB
1.2	Koordinator	3.7C	200MB	3C	100MB
2.1	Default	3.7C	200MB	0.2C	100MB
2.2	Koordinator	3.7C	200MB	0.2C	100MB

Table 2: Application Workload Configuration

No.	Scheduler	CPU Req.	Mem Req.	Desire Pod Count (Driver/Executor)	Job Type
1.1	Default	1	512 MB	2 (1/1)	SparkPI
1.2	Koordinator	1	512 MB	2 (1/1)	SparkPI
2.1	Default	1	512 MB	5 (1/4)	SparkTC
2.2	Koordinator	1	512 MB	5 (1/4)	SparkTC

Table 3: Spark Workload Configuration

Figure 3 illustrates the results for the second set of experiments. The flat horizontal dashline represents CPU and memory requests for the application workload, while the curved dashlines represent cluster CPU and memory usage using the default scheduler. The difference between the red horizontal dashline and the curved dashline represents the potentially reusable resources. The solid lines represent cluster usage after switching to Koordinator.

We observed a significant increase in overall cluster usage. Notably, in experiment 2.2, although the Spark application ends successfully, only 2-3 requested executors are scheduled. This limitation is due to the application's headroom, which is nearly 1.2C (35% * 3.7), preventing more executors from being colocated.

No.	Scheduler	Spark Workload Result
1.1	Default	Rejected
1.2	Koordinator	Finished
2.1	Default	Rejected
2.2	Koordinator	Finished*

Table 4: Experiments Result

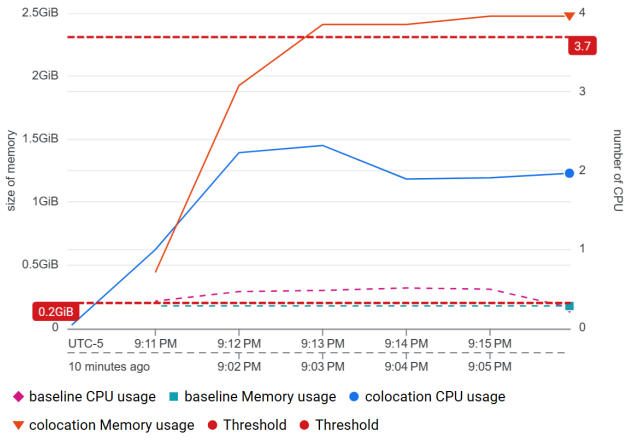

Figure 3: Result for 2nd Set of Experiments

5 Related Work

Kube-Scheduler default scheduler for Kubernetes, supports scheduling based on resource requirements, Pod and Job correspond to different types of workloads.

Koordinator QoS-based scheduling for efficient orchestration of microservices, AI, and big data workloads on Kubernetes[12]

Kube-Batch a batch scheduler for Kubernetes, providing mechanisms for applications which would like to run batch jobs leveraging Kubernetes. [7]

Bytedance Co-locate Practice with Intel Performance Evaluation and Optimization for Workload Colocation[2]

6 Future Work

This project has explored the co-location of Spark batch jobs alongside simulated workloads. Moving forward, it is necessary to expand the scope of experimentation to include simulations of real-life workload data, such as those from Microsoft Philly traces [6], and to validate these simulations with actual applications. Additionally, the scope should be broadened to incorporate a more diverse range of workloads to thoroughly assess the benefits and effectiveness of over-committing resources. Lastly, evaluating this strategy in larger-scale systems will provide valuable insights, as these systems offer greater potential for over-committing services.

7 Resources

- Project Repository
- Project Poster

8 Contribution

Mondo Enviroment Setup, Kubernetes Configuratioon and Workload Dispatch, System Design, Evluation.

Zihao Application Workload Simulation, Observability Data Query and Figure Generation, Intro, Background and Future Work.

References

[1] Trends for DevOps engineering in 2023, Jan. 2023.

[2] I. ByteDance. Bytedance: Performance evaluation and optimization for workload colocation. https://www.intel.com.br/content/dam/www/public/us/en/documents/white-papers/Intel_ByteDance_WhitePaper.pdf.

[3] flow2000. Looksbusy. https://github.com/flow2000/lookbusy.

[4] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[6] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads, Aug. 2019. arXiv:1901.05758 [cs].

[7] Kubernetes. kube-batch: a batch scheduler for kubernetes. https://github.com/kubernetes-retired/kube-batch.

[8] Kubernetes. Kubernetes controller. https://kubernetes.io/docs/concepts/architecture/controller/.

[9] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. {Heterogeneity-Aware} Cluster Scheduling Policies for Deep Learning Workloads. pages 481–498, 2020.

[10] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[11] A. Spark. Spark examples. https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples.

[12] K. Team. Koordinator, qos-based scheduling for efficient orchestration of microservices, ai, and big data workloads on kubernetes. https://koordinator.sh.

[13] terraform. Terraform. https://www.terraform.io/.

[14] volcano. volcano. https://volcano.sh/en/.