Towards Efficient Microservices Management Through Opportunistic Resource Reduction

Md Rajib Hossen
Dept of Computer Science and Engineering
The University of Texas at Arlington
Arlington, TX, USA
Email: mdrajib.hossen@mavs.uta.edu

Mohammad A. Islam
Dept of Computer Science and Engineering
The University of Texas at Arlington
Arlington, TX, USA
Email: mislam@uta.edu

***Abstract*—Cloud applications are moving towards microservice-based implementations where larger applications are broken into lighter-weight and loosely-coupled small services. Microservices offer significant benefits over monolithic applications as they are more easily deployable, highly scalable, and easy to update. However, resource management for microservices is challenging due to their number and complex interactions. Existing approaches either cannot capture the microservice inter-dependence or require extensive training data for their models and intentionally cause service level objective violations. In our work, we are developing a lightweight learning-based resource manager for microservices that does not require extensive data and avoid causing service level objective violation during learning. We start with ample resource allocation for microservices and identify resource reduction opportunities to gradually decrease the resource to efficient allocation. We demonstrate the main challenges in microservice resource allocation using three prototype applications and show preliminary results to support our design intuition.**

Index Terms*— *Microservices; Resource-Management; Cloud-Computing; Service-Level-Objective; Kubernetes.

I. Introduction

Cloud applications have been evolving from monolithic architecture to microservice architecture. For instance, leading IT companies, such as Netflix, Amazon, eBay, Spotify, and Uber are adopting microservices [1]–[3]. In contrast to monolithic applications with a few large layers [4], microservice architectural style consists of a set of loosely coupled small-scale services deployed independently, each with its process and database. The services communicate via lightweight communication mechanisms, such as HTTP API, gRpc [2], [5]–[8]. Figure 1 shows the difference between monolithic and microservice architectures. Compared to monolithic applications, microservices are more easily deployable, highly scalable, easy to update components, and have better fault tolerance.

Microservices, however, come with their own sets of challenges. As shown in Figure 1, microservices have complex communication between them. To complete a request, a microservice may call one or several other microservices in parallel or sequential order. Due to these complicated relationships, microservice resource management becomes challenging. Naive approaches may waste resources by over-provisioning or violate the Quality of Service (QoS) by underprovisioning. Moreover, current solutions for resource management of clouds are for monolithic applications [9]–[14]. Although they provide excellent solutions for existing monolithic applications and data centers, they fail to capture the complicated relationship among microservices. As a result, these approaches can not be applied directly to microservices.

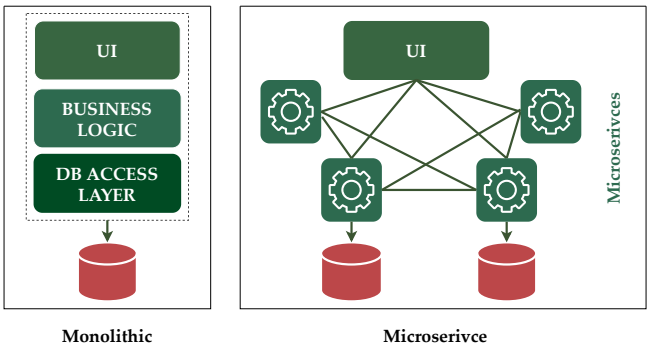


Fig. 1. Architecture of Monolithic and Microservices. Monothilic application blocks are defined and deployed as single units. Microservices offer more flexibility by decoupling an application into several small units and deploying independently.

With the need for efficient resource management systems, research in resource management for microservices is gaining momentum in academia [15]–[21]. Prior works focusing on adopting popular approaches from monolithic applications such as allocation rules and model-based resource management fail to capture microservice inter-dependences and interactions in scalable fashion [15], [16]. Meanwhile, Machine Learning (ML) based approaches require extensive training data for building reliable models [17]–[19]. Their heavy dependency on data makes them slow to adapt to changes in microservice deployment such as software updates and hardware changes, even to changes in resource demand due to change workload intensity. More importantly, ML based approaches need to create Service Level Objective (SLO) violations to train the models intentionally [18], [20].

In this paper, we present our preliminary results towards developing a lighter-weight resource manager for microservice that learn efficient resource allocation through iterative interaction with the microservice application, yet do not rely on extensive data and avoid intentional SLO violation

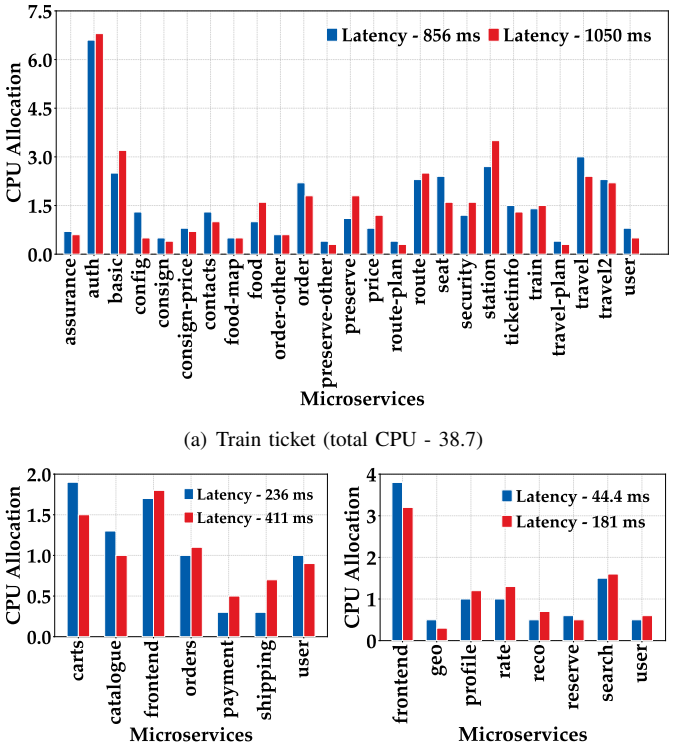
during the learning process. As a work-in-progress work, we show the key challenges of efficient resource allocation using three prototype microservice implementations. We then present our solution approach where we start with sufficient resource allocation for every microservice and then identify resource reduction opportunities to allocate efficiently. We avoid causing SLO violations during the learning since we always maintain at least more resources than required for each microservice. We also present experimental results from the prototype applications supporting our solution intuition. Finally, in future work, we discuss the technical challenges in our approach and our plans to address them.

The rest of the paper is organized as follows. We discuss related works in Section II. In Section III, we introduce the problem of microservice resource management and its challenges. We present our proposed solution in Section IV followed by concluding remarks and directions for future work in Section V.

II. Related Work

Existing efforts on microservice resource management can be categorized as heuristics-based, model-based, and machine learning-based approaches. Kubernetes [22] is a popular container orchestration system that scales container resources horizontally and vertically. It decides resource allocation by defining thresholds for performance metrics such as CPU Utilization and Memory Utilization [23]. Kwan et al. [15] propose a rule-based solution to autoscale resources based on CPU and memory utilization. These rule-based systems require application profiling for identifying the threshold values that vary from application to application and require frequent updates with system changes. To ease these efforts, several other studies model-based resource management systems such as queue network, and application profiling based approach to find optimal resources [16].

ML is another heavily used approach in resource management for microservices. ML approaches can be divided into 1) data-driven and 2) Reinforcement Learning approaches. Data-Driven approaches usually work by finding the relationship between performance metrics and response time [17]–[19]. Jindal et al. [17] collected data to calculate the serving capacity of microservices without violating SLO. Yu et al. [19] proposed an approach to identify scaling needed services by using the ratio of 50 percentile and 90 percentile response time and used Bayesian optimization to scale them horizontally. Zhang et al. [18] proposed a space exploration algorithm to gather data to train two machine learning models - Convolutional Neural Network and Boosted Trees. These models are then used to generate resources for various workloads. Although these papers consider microservice complexity when deciding resources, they need extensive experimental data and intentional SLO violations to train the models. For example, Zhang et al. [18] collected performance data after intentionally causing response time going 20% above the SLO. Such intentional SLO violations may not be feasible for production systems. Reinforcement Learning or Online Learning is used to manage resources dynamically for microservices [20], [21]. Qiu et al. [20] first identifies bottleneck microservices using a support vector machine and then uses reinforcement learning to provide resources for bottleneck microservices. However, they injected anomalies into the system to generate training data which may not be possible in real life. The AlphaR [21] uses a bipartite graph neural network to determine the application characteristics and then uses reinforcement learning to generate resource management policy. Although reinforcement learning provides online learning, they can suffers from a long training time.



(a) Train ticket (total CPU - 38.7)

(b) Sock Shop (total CPU - 7.5)

(c) Hotel reservation (total CPU - 9.4)

Fig. 2. Impact of resource distribution among microservice in response time. The X-axis shows the microservices in each applications, and the Y-axis shows the CPU allocation of each microservices.

III. Overview

In this Section, we formalize the problem statement, discuss the challenges in finding efficient resources, and describe the experimental settings for implementations.

A. Problem Statement

Using a discrete time-slotted model indexed by k where the resource management decisions are refreshed once every time slot, we formalize the microservice management as the following optimization problem, Efficient Microservice Management (EMM) where the objective is to minimize the

total resource allocation with the performance satisfying the SLO.

$$\text{EMM: minimize}_{\mathbf{r}(k)} \quad \sum_{n=1}^{N} r_n(k) \tag{1}$$

$$\text{subject to} \quad \mathcal{L}(\mathbf{r}(k)) \leq SLO, \tag{2}$$

Here, N is the number of microservices, $\mathbf{r}(k) = (r_1(k), r_2(k), \cdots, r_N(k))$ is the resource allocation, and $\mathcal{L}(\mathbf{r}(k))$ is the performance of the microservice application which is a function of the resource allocation vector $\mathbf{r}(k)$. We define the microservice performance (i.e., $\mathcal{L}(\mathbf{r}(k))$) as the end-to-end 95^{th} percentile response time.

B. Prototype Applications

To study EMM, we deploy three benchmark microservice applications - a ticket booking platform *Train Ticket* [24], a reservation application *Hotel Reservation* [8], and a e-commerce website *Sock Shop* [25]. The *Train Ticket*, *Sock Shop*, and *Hotel Reservation* applications consist of 41, 13, and 18 microservices, respectively. We deploy these applications on a Kubernetes [22] cluster with three nodes, each with two 20-cores Intel Xeon 4210R processors. In our resource allocation problem, we mainly consider CPU resource allocation for each microservice. As memory allocation cannot be easily changed without restarting the containers, we allocate enough memory to each container to ensure that the memory does not become the bottleneck resource. We can set the memory and the initial CPU allocation following offline profiling used in typical cloud applications [26]. Notably, in cloud deployments, it is a common practice to overprovision (e.g., allocate 20% more resource than required), which fits perfectly with our solution approach.. We also use Prometheus [27] and Linkerd [28] to collect performance metrics from containers and the applications. For all of our prototype implementations, we consider 95^{th} percentile end-to-end response time as the performance metrics.

C. Challenges in EMM

Solving EMM is particularly difficult for microservices because it is very hard to accurately estimate $\mathcal{L}(\mathbf{r}(k))$ in practice. The main reasons are that microservices are interdependent, and the behavior of microservices changes based on the workloads and CPU allocations. Moreover, each request to the applications needs to be processed by several microservices. Hence, all the microservices in the execution path dictate a request's end-to-end response time. If one microservice becomes a bottleneck, the end-to-end response time will increase. As a result, even with a fixed total CPU allocation, different CPU distributions (i.e., different $\mathbf{r}(k)$) among microservices may produce widely different response times. Figure 2 shows a motivating example, where we see that for the same amount of CPU allocation, the response time varies significantly when the resource distribution among microservices change.

In Figure 2(a), we see that the response time for train ticket increases more than 20% with an unfavorable allocation. In Figure 2(b), sock shop shows 74% increase and in Figure 2(c), hotel reservation shows more than 300% increase in response time due to change in resource distribution.

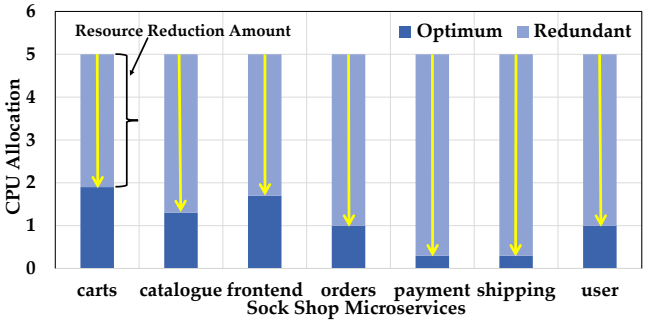


Fig. 3. Illustration of opportunistic resource reduction-based approach.

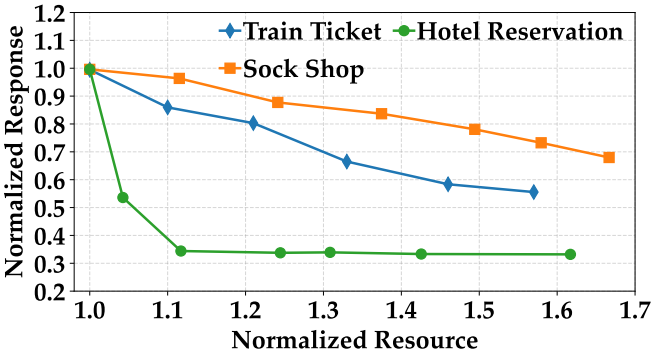


Fig. 4. Changes in response time with resource allocation demonstrating our intuition for opportunistic resource reduction-based approach.

IV. Opportunistic Resource Reduction

To circumvent the challenge of estimating the response time $\mathcal{L}(\mathbf{r}(k))$, we adopt a feedback-based approach where we do not need to know $\mathcal{L}(\mathbf{r}(k))$, instead we use the feedback from the application to find the response time for a given resource allocation. We collect this performance feedback at the end of a resource allocation time slot. We then can find the minimum resource allocation that satisfies the SLO by iteratively interacting (i.e., allocating resource and tracking its impact) with the application. This iterative interaction can be interpreted as a learning-based solution where we learn the efficient resource allocations.

However, since we are using a live system for the learning, we need to carefully decide our resource allocation in each iteration to avoid SLO violations. Hence, instead of finding efficient resource allocation, we reframe our solution approach to finding resource reduction opportunities. More specifically, we deploy our applications with sufficient resources to satisfy the SLO and then identify *opportunities for resource reduction* from different microservices based on the application's performance. Figure 3 illustrates our approach on sock shop application where we start with CPU allocation of "5" for each microservice and gradually reduce the resources in iterations

to reach the optimum allocation. Since we start from a high resource, no microservice becomes the bottleneck to cause SLO violation during this iterative resource reduction.

To corroborate the intuition of our approach, we run preliminary experiments on our microservice implementations. Our goal in these experiments is to show that a gradual decrease in the resource can lead to the optimum allocation. We first find the optimum resource allocation for each application by exhaustive trials and errors. We run each application several times, and during each new run, we increase the resource of a few randomly selected microservices. The response time of these experiments is shown in Figure 4 where we normalize each application's response time to their respective SLO and resource allocation to their respective optimum. We see that a gradual decrease in the resources leads the response time closer to the SLO. This results support our resource reduction-based design intuition. Moreover, it proves that, we can reach the optimum resource allocation eventually.

Note that, the trail and error approach is not applicable in a live system due to its impact on the QoS. Finally, in the above experiment that we randomly increase the resource instead of resource reduction as we have yet to develop our resource reduction algorithm. Nonetheless, the observation of the evolution of the response time with resource allocation changes in these experiments holds for the reduction algorithm.

V. Conclusion

This paper is part of our ongoing project of developing a complete resource manager that finds the optimum resources for every application without human intervention and without degrading the Quality of Services (QoS).

The preliminary results demonstrate the potential of opportunistic resource reduction. To identify the resource reduction opportunity, we plan to use the difference between the response time and the SLO. This is because, in general, resource reduction increases in response time. Hence, any room for response time increase (i.e., the difference between response time and SLO) can be interpreted as a resource reduction opportunity. The response time is an application-level metric and does not reveal which microservices are the best candidates for resource reduction. Hence, we plan to incorporate microservice-level performance metrics in our resource reduction algorithm. We will follow two principles to avoid SLO violations for future iterations. First, we will maintain microservice-level performance (e.g., utilization) upper limits for SLO compliance. We will refrain from further resource reduction if current metrics exceed the upper limit. The limits will be dynamically updated based on SLO satisfaction. Second, we will be conservative in resource reduction. Instead of reducing a considerable amount of resources in one iteration, we will use a gradient descent approach to reduce a small number of resources and then examine the system performances. This way, it will be guaranteed that even if the system can not find exact optimum resources, it will not violate SLO. In addition to avoiding SLO violations, we will also incorporate workload changes to the optimum resource allocation as the workload intensity directly affects the resource requirement for satisfying SLO.

References

[1] "Mastering chaos: A netflix guide to microservices," https://www.infoq.com/presentations/netflix-chaos-microservices/, accessed: 01/07/2022.

[2] "Introduction to microservices," https://www.nginx.com/blog/introduction-to-microservices, accessed: 01/05/2022.

[3] "Leading companies embracing microservices," https://www.divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others, accessed: 01/08/2022.

[4] "The definition of monolithic," https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/, accessed: 01/05/2022.

[5] "The definition of microservice," https://martinfowler.com/microservices/, accessed: 01/05/2022.

[6] "What are microservices?" https://microservices.io/, accessed: 01/05/2022.

[7] "Microservice vs monolithic architectures," https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/, accessed: 01/05/2022.

[8] Y. G. et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *ASPLOS*, 2019.

[9] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," in *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018, pp. 1–33.

[10] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *ACM SIGOPS 22nd SOSP*, 2009, p. 261–276.

[11] "Hadoop fair scheduler," https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html, last Accessed: 01/05/2022.

[12] "Amazon aws autoscale," https://docs.aws.amazon.com/autoscaling/index.html, last Accessed: 10/05/2021.

[13] "Azure autoscale," https://azure.microsoft.com/en-us/features/autoscale/, last Accessed: 10/05/2021.

[14] M. Wajahat, A. A. Karve, A. Kochut, and A. Gandhi, "Mlscale: A machine learning based application-agnostic autoscaler," *Sustain. Comput. Informatics Syst.*, vol. 22, pp. 287–299, 2019.

[15] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *2019 IEEE 39th ICDCS*, 2019, pp. 80–90.

[16] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," *Proceedings - International Conference on Distributed Computing Systems*, vol. 2019-July, pp. 1994–2004, 2019.

[17] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *ACM/SPEC ICPE*, 2019, p. 25–32.

[18] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *International Conference on ASPLOS*, 2021, pp. 167–181.

[19] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," *ICWS 2019 - Part of the 2019 IEEE World Congress on Services*, pp. 68–75, 2019.

[20] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on OSDI*, Nov. 2020, pp. 805–825.

[21] X. H. et al., "Alphar: Learning-powered resource management for irregular, dynamic microservice graph," in *2021 IEEE IPDPS*, 2021, pp. 797–806.

[22] "Kubernetes: Production grade container orchestration," https://kubernetes.io/, accessed: 01/20/2022.

[23] "Kubernetes horizontal pod autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, last Accessed: 10/05/2021.

[24] Z. et al., "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[25] "Sock shop microservice demo," https://microservices-demo.github.io/, accessed: 08/31/2021.

[26] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ml-driven performance debugging in microservices," in *26th ACM International Conference on ASPLOS*, 2021, p. 135–151.
[27] "Prometheus - from metrics to insights," https://prometheus.io/, last Accessed: 10/08/2021.
[28] "Linkerd: A different kind of service mesh," https://linkerd.io/, accessed: 08/31/2021.