RAMBO: Resource Allocation for Microservices Using Bayesian Optimization

Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis

Abstract—Microservices are becoming the defining paradigm of cloud applications, which raises urgent challenges for efficient datacenter management. Guaranteeing end-to-end Service Level Agreement (SLA) while optimizing resource allocation is critical to both cloud service providers and users. However, one application may contain hundreds of microservices, which constitute an enormous search space that is unfeasible to explore exhaustively. Thus, we propose RAMBO, an SLA-aware framework for microservices that leverages multi-objective Bayesian Optimization (BO) to allocate resources and meet performance/cost goals. Experiments conducted on a real microservice workload demonstrate that RAMBO can correctly characterize each microservice and efficiently discover Pareto-optimal solutions. We envision that the proposed methodology and results will benefit future resource planning, cluster orchestration, and job scheduling.

Index Terms—Distributed applications, emerging technologies

1 Introduction

The microservice architecture is becoming the leading trend for cloud applications. According to a survey in 2020 [1], 52 percent of respondent companies use microservices. Compared to having a monolithic complex application, the microservice architecture divides an application into loosely coupled tiers. It enables modular and composable software development, and opens the opportunity for elastic scaling. However, it also raises urgent challenges for efficient datacenter management.

Guaranteeing end-to-end Service Level Agreement (SLA) while optimizing resource allocation is critical for both cloud service providers (CSPs) and users. While the SLA is essential for performance, customer satisfaction, and company reputation, resource allocation is a critical contribution to the total cost of ownership (TCO) of the infrastructure. These two objectives establish a range of tradeoffs between performance and cost. For example, if we reduce the TCO by allocating fewer resources, the tail latency effects and SLA violations may increase [2]. To achieve the best performance/cost tradeoffs, one needs to correctly characterize resource requirements and decide the allocation for each microservice, such as the number of replicas, CPU cores assigned for execution, cache ways, and the amount of allocated memory bandwidth. However, one application may contain hundreds of diverse microservices and have complex dependencies across tiers, forming an enormous space of possible resource configurations that is unfeasible to explore exhaustively or manually.

Since SLA compliance usually has higher priority over cost, a common approach today is to overprovision resources [3], which accounts for the worst-case execution scenario. Overprovisioning negatively impacts on cost as it leaves a large portion of resources underutilized for most of the time. Prior approaches that leverage autoscaling either rely on heuristics [3] to scale each microservice individually, which ignores dependencies and is suboptimal, or require collecting a massive training dataset [4], [5], which could be unfeasible in a general scenario. Moreover, existing methods often ignore the joint management of shared resources such as last-level cache (LLC) or memory bandwidth. There is an urgent need for an automated, sample-efficient method to allocate resources for microservices with a holistic view of their diverse characteristics and dependencies, and considers the two objectives of performance and cost simultaneously.

Prior work [6], [7] has leveraged multi-objective evolutionary methods on other datacenter workloads and management. Recent work has shown benefits and faster convergence of BO compared to alternative techniques such as evolutionary algorithms [8]. However, most of the practice today is on single-objective BO and monolithic workloads, with multi-objective BO and microservices still being open research problems.

In this paper, we introduce RAMBO, a general and efficient framework for SLA-aware and TCO-optimized resource allocation for microservices. We formulate and solve a multi-objective optimization problem that considers both SLA and the amount of resources needed to satisfy the SLA. Specifically, we demonstrate novel practical use of multi-objective BO for tuning microservices applications, where both performance and cost are critical and should be optimized simultaneously. Also, we show that for the microservices we evaluated, BO has advantages over evolutionary techniques, in our case a Genetic Algorithm (GA), described in more details in Section 3.3. RAMBO requires no microservices tracing or instrumentation.

The use of Bayesian Optimization delivers several key benefits for microservice applications: (i) *Sample efficiency*: BO obtains Pareto-optimal configurations with a limited number of trials; (ii) *Automated relevance determination*: BO automatically captures dependencies and characteristics of microservices; (iii) *Exploration-exploitation*: BO smartly balances local and global search; (iv) *Multi-objective*: we extended standard BO to account for multiple objectives simultaneously, thus spanning the full range of performance/cost tradeoffs.

Our experiments with a real microservice workload demonstrate that RAMBO can correctly characterize each microservice and explore the dependencies between microservices, and improve performance by up to 6.3× with minimal resource allocation. To further showcase its benefits, we compare the BO algorithm against random sampling and a multi-objective GA and show its superiority. We envision that our technique and results can help practitioners to improve capacity planning, orchestration, and scheduling.

2 Motivation and Challenges

Overprovisioning. To maintain strict SLA compliance of cloud applications, CSPs often overprovision resources to account for tail latencies, which leads to a dramatic waste of resources. With overprovisioning, the datacenter capital spending increases since more machines and network resources are needed to support a given workload. The associated space, power, and cooling expenses contribute even further to the TCO [9]. With the rise of microservices, overprovisioning is no longer a practical solution since the waste of resources would accumulate along the chain of services, escalating the TCO.

Large and Complex Search Space. Each application may contain tens to hundreds of microservices. One needs to consider multiple types of resources for each microservice, such as the number of replicas, cores, memory, and cache. If we consider n service containers and m possible core allocation values per container, the

- *Qian Li and Christos Kozyrakis are with the Stanford University, Stanford, CA 94305 USA. E-mail: qianli@cs.stanford.edu, kozyraki@stanford.edu.*
- *Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, and Michael Kishinevsky are with the Intel Corporation, Hillsboro, OR 97124 USA. E-mail: {bin.li, pietro.mercati, ramesh.g.illikkal, charlie.tai, michael.kishinevsky}@intel.com.*

Manuscript received 10 Feb. 2021; accepted 6 Mar. 2021. Date of publication 17 Mar. 2021; date of current version 1 Apr. 2021.
(Corresponding author: Qian Li.)
Digital Object Identifier no. 10.1109/LCA.2021.3066142

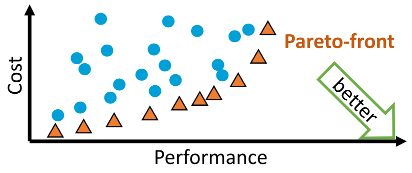

Fig. 1. An example of two-objective space and Pareto-front.

search space would be m^n, i.e., it grows exponentially with the number of service containers. Thus, the space is prohibitively large for exhaustive search, and unfeasible to navigate manually, due to the large number of knobs and the wide option ranges. Moreover, as suggested in [5], one needs to tune all knobs with a holistic view due to the complex dependencies between microservices. As request rate changes, one needs to consider both scale-out and scale-up decisions for all microservices that are on the critical path at the same time. Otherwise, a single bottleneck can cause SLA violations, or a suboptimal configuration could lead to high TCO.

Resource Usage versus Resource Requirement. Existing resource allocation methods usually rely on resource monitoring data [3]. However, we notice that resource usage does not always correlate to resource requirements. For example, the Stream [10] workload occupies a large amount of LLC but is not sensitive to cache sizes, thus allocating fewer LLC ways could still maintain good performance. The unassigned cache ways can be used for other co-located processes on the same server and further improve resource efficiency. Unfortunately, monitoring based allocation/autoscaling techniques cannot discover such opportunities. Similarly, allocating more CPU cores may not always improve performance because high CPU utilization may not correlate to SLA violations [2], or the application has a limited number of threads [4].

In summary, we need an automated, sample-efficient method to characterize and explore optimal resource allocation configurations for microservices correctly.

3 RAMBO Framework

We first formulate resource allocation as a multi-objective optimization problem. Then, we describe how to apply BO to solve the problem. Finally, we discuss the architecture of RAMBO.

3.1 Multi-Objective Optimization Problem

In this paper, the goal of optimization is to find the best configuration of resources to maximize the performance of a microservice application while minimizing resource cost, which is a bi-objective problem. As shown in Fig. 1, if we depict the two objectives as axis on a plane, the configurations delivering the best tradeoffs between performance and cost stretch towards the lower-right corner. Such configurations are called "Pareto-optimal" and together they form the "Pareto-front".

Assuming that an application has n microservices, a configuration consists of the following knobs for each microservice $i \in [0, 1, \ldots, n-1]$: number of replicas x_i^{replica}, number of assigned CPU cores per replica x_i^{cpu}, allocated memory bandwidth x_i^{membw} and number of allocated LLC ways x_i^{llc}. We define the performance metric as the maximum throughput under latency constraint. To obtain the metric, we send various loads (request rates) to the application and measure the 99th percentile latency and throughput, then record the maximum achievable throughput that satisfies the latency constraint. This is a common approach to measure performance for cloud applications. To measure resource cost, we define a cost function which is a weighted sum of three components

$$\text{cost}(x) = \alpha \frac{\sum_i x_i^{\text{cpu}} \cdot x_i^{\text{replica}}}{\text{total CPU}} + \beta \frac{\sum_i x_i^{\text{llc}}}{\text{total LLC}} + \gamma \frac{\sum_i x_i^{\text{membw}}}{\text{total MemBW}},$$

where weights α, β, γ represent the relative importance of these components and should be configured by the user. In our case, we assigned $\alpha = 2$, $\beta = 1$, $\gamma = 1$, which means that we give twice more importance to computational resources (cpu, replica) compared to memory (membw, llc). Each component, multiplied by these weights, is a ratio ranging from 0 (resource is idle) to 1 (resource is fully allocated). Note that a computational ratio greater than 1 means the CPU is oversubscribed, and we will timeshare CPUs across applications. For example, if all resources are fully allocated, then $\text{cost} = 2*1+1*1+1*1 = 4$. If half of the total CPUs are used, then $\text{cost} = 2*0.5+\ldots = 3$. We use this cost function since it is easy to interpret and customize, but a different form can be used as long as it reasonably represents resource cost for microservices (e.g., TCO).

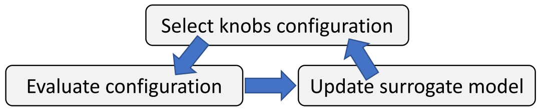

Fig. 2. Bayesian optimization overview.

3.2 Bayesian Optimization Overview

Bayesian Optimization (BO) is a state-of-the-art technique for black-box optimization that iteratively updates a surrogate model of the target objective and uses it to guide the search for promising solutions. BO automatically balances local search in good regions of the input space and global search in unexplored regions. Moreover, BO can handle multiple objectives, thus we can apply it to our bi-objective problem. First, we pre-train the surrogate model with a dataset collected randomly. Then, BO starts to explore the configurations as shown in Fig. 2.

At each iteration, BO selects the next configuration to evaluate by maximizing the "acquisition function" and returning its argument. Such acquisition function takes model predictions for unseen configurations as input, and estimates the amount of improvement the unseen ones can achieve over the current best result. This function is designed to balance both cases: configurations that are close to the current best ones (local search), and configurations that are very different from anything seen thus far (global search). Maximizing it means to select the most promising configuration for evaluation. Then, BO evaluates the selected configuration on the target system, and collects the corresponding objective metrics. Finally, this newly obtained data point, with input knobs configuration and output objective metrics, is used to update the surrogate model and the next iteration can start. The algorithm usually stops either when running out of iterations, or with some rule on the quality of results, such as stopping if no better solution is discovered in 10 consecutive iterations.

To use BO, we need to specify an acquisition function and a surrogate model to employ in the optimization loop. We adopt the SMSego acquisition function, which enables multi-objective optimization and has been shown in previous work [11] to have performance comparable to other advanced acquisition functions (e.g., entropy based), while being more computationally efficient. For the surrogate model, we choose a single-output Gaussian Process (GP) [12] for each objective. GP achieves good accuracy with a relatively small number of training points, which in turn helps the BO to converge quickly. Also, GP has convenient mathematical properties that allow to train it and tune its hyperparameters in closed form.

Moreover, we need to specify a "kernel" for the GP model. The kernel would determine the family of functions that the GP is able to model. We choose the Matern52 kernel, which is a common choice for modeling realistic processes with a low degree of smoothness. GP training time has cubic time complexity on training set size, but this is not a problem for us, since our application typically needs tens/hundreds of data points. While in this work

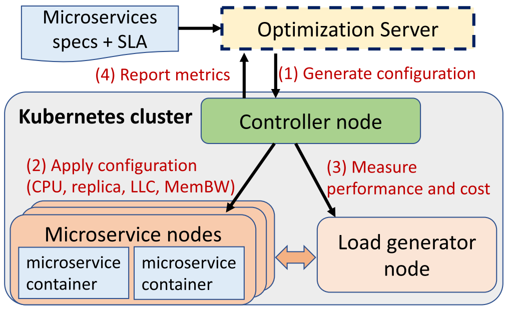

Fig. 3. RAMBO architecture. Workflow steps are marked with numbers.

we focus on sequential BO (i.e., selecting and evaluating one configuration at a time), our framework can also support selecting a batch of candidate solutions for parallel evaluation when multiple systems are available.

We also considered alternative algorithms for comparison (see Section 4.2): random search and Genetic Algorithm (GA). GA is a commonly used algorithm for optimization problems, which generates new configurations through crossover and mutation of the best historical data points. While simpler to implement, GA does not build a surrogate model and thus has generally slower convergence compared to BO.

3.3 Software Stack

Fig. 3 shows RAMBO with Kubernetes [13]. A user only needs to submit microservice specifications (e.g., YAML files) and SLA requirements to the system, and after that the tuning process is fully automated. For load generation, we leverage HTTP/gRPC load generation tools [14]. For CPU and replica allocation, we leverage the Kubernetes replica set feature. As for LLC and memory bandwidth allocation, we use Intel® Resource Director Technology (RDT) [15], including Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA).

RAMBO provides multiple plugins for optimization algorithms. In this work, we use BO, GA, and random search, which are all implemented in Python. We implement GP with the *GPy* library. Our Genetic Algorithm is an in-house implementation inspired by NSGA-II, a common multi-objective GA [16].

The workflow at each optimization iteration is as follows (marked with reference numbers in Fig. 3): (1) the optimization server generates a new configuration for microservices based on BO and sends it to the controller node; (2) the cluster applies the configuration, including replica, CPU, LLC, and MemBW allocation; (3) the controller node interacts with the load generator node that measures the maximum throughput under end-to-end SLA constraints for the deployed microservices, and calculates the cost; (4) the controller node reports objective metrics back to the optimization server, and BO will update the surrogate model and start the next iteration.

4 Evaluation

Cluster Setup. We deploy our framework on a local five-node Kubernetes cluster, with three workers, one controller, and one load generator. We allocate the controller node and the load generator node on two separate dedicated machines, and run microservices on the remaining three servers. Each machine has an Intel Xeon Platinum 8176 CPU @ 2.5 GHz, which is dual-socket, with 28 cores per socket and 11 shared LLC ways. To minimize interference with the workload execution, we deploy the optimization server on a dedicated machine outside of the Kubernetes cluster. The servers are connected with a 100 Gbps top-level switch so that applications are not constrained by network capacity.

Note that our results illustrate the optimization potential rather than the performance of the Skylake server systems.

4.1 RAMBO for Realistic Workload

We evaluate our approach with the “read-home-timeline” query from the DeathStarBench SocialNet workload [2], a popular realistic benchmark for microservices. It consists of 29 microservices and implements a media social network. We pre-populate the user database with a social graph based on Reed98 Facebook Networks [17], with 962 users and 18.8K graph edges. Then, we compose 150 posts per user, with 512 bytes per post. We use *wrk* [14], an HTTP load generator, to send requests that randomly select a user and read 10 consecutive posts. We set the P99 latency constraint to 50 ms.

This workload has 34 knobs in total: number of CPUs and Replica for each tunable microservice, plus shared LLC and MemBW knobs. For this experiment, we exclude persistent database services, and we keep Memcached replica to 1; thus we have 18 out of 29 microservices that are tunable. Also, we place all microservices into a single RDT group; thus the same LLC and MemBW knobs are applied to all microservices and they are only counted once. We leave how to divide services into separate RDT groups as future work. We set LLC range to 2–11 ways with step 1 (baseline value 11), the MemBW range to 20–100 percent with step 10 percent (baseline value 100 percent), the CPU range to 1–8 with step 1 (baseline value 4 for services on the critical path, otherwise 1), and the Replica range to 1–4 with step 1 (baseline value 1). The baseline is tuned manually by experts, which is a reasonable starting point. The search space size is 4.35×10^{26}, which is unfeasible to search exhaustively. We configure BO to run for 150 iterations, including 40 initial random exploration samples. For each evaluation, we repeat the measurement 3 times and report the median value for stable results. Each evaluation phase takes approximately 7 minutes.

The results of RAMBO are shown in Fig. 4a, where we highlight data regions representative of different performance/cost tradeoffs on the Pareto-front. In particular, the “High performance” point achieved 6.3x higher throughput while only using 20 percent more resources compared to the baseline configuration. The balanced points also achieve higher throughput and use lower resources

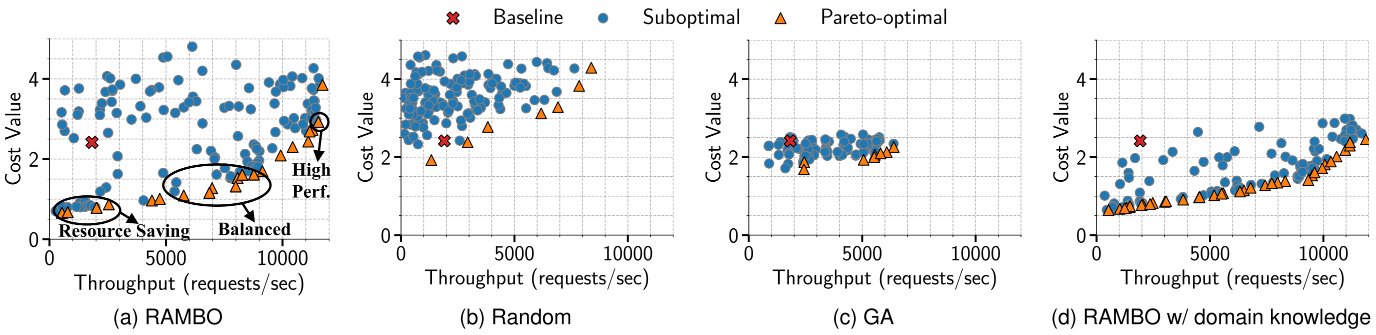

Fig. 4. Performance and cost for all iterations of SocialNet workload.

TABLE 1
Representative Configurations From SocialNet Pareto-Front

RPS	Nginx		Home-timeline		Post-storage		Mem$	LLC	MemBW
	cpu	replica	cpu	replica	cpu	replica	cpu		
532	2	1	1	1	1	1	1	2	20
3068	4	3	4	1	4	1	1	2	20
6320	7	4	5	1	5	1	1	2	20
11166	8	4	5	1	8	2	2	7	70

than the baseline. Interestingly, RAMBO also explores the "Resource saving" area where it uses minimum resources, which is useful when the throughput requirement is low. The suboptimal configurations require higher resources to achieve the same throughput as the Pareto optimal ones, where we can save up to $5\times$ resources to achieve the same throughput.

We examine the generated configurations and report representative data points in Table 1. We validate that RAMBO can automatically discover the lower LLC and MemBW requirement of SocialNet at low load. It also discovers that higher LLC and MemBW are needed for high loads. RAMBO explores balanced configurations across all microservices and resource types simultaneously, and it correctly characterizes each microservice and correlates performance with resource allocation with a holistic view. This is a major benefit compared to prior autoscaling mechanisms that scale each microservice independently, which might not find a good end-to-end balance.

4.2 Algorithms Comparison

We compare BO with random search (Fig. 4b) and GA (Fig. 4c), with the same number of total iterations. Because random search explores without guidance, it only examines a suboptimal region compared to other algorithms, due to the large search space. The GA algorithm is only able to explore a limited region of the search space, thus resulting slower and less effective than BO. GA may find a better Pareto-front if we allow it to run more iterations than BO, incorporate more advanced techniques [7], [18], and carefully craft for this specific problem.

4.3 Domain-Specific Knowledge

In Fig. 4d, we further demonstrate that domain-specific knowledge can guide BO to achieve even better results. By checking the critical path of the query, we configure BO to only tune nine critical knobs, while keeping other knobs to the minimum value. In the high performance region, the result indicates that domain-specific knowledge can help achieve even higher throughput while further reducing resource allocation (cost value). This observation motivates us to investigate techniques that allow BO to automatically prioritize critical knobs and further improve sample efficiency as future work.

5 Conclusion and Future Work

This paper presents RAMBO, an automated resource allocation framework for microservices using Bayesian Optimization. With a realistic workload, we showed that our method can correctly characterize each microservice and efficiently explore the Pareto-front of minimum resource allocation and maximum throughput under SLA constraints.

Our results can be used to guide resource planning, cluster orchestration, and scheduling. Before deploying a microservice application, a resource planner can use RAMBO to tune configurations for different performance/cost tradeoffs (e.g., "balance" versus "high performance"). These configurations can then be loaded onto a runtime orchestrator, which would decide when to switch based on the total request rate or the time of the day. We understand that the variability and unpredictability of some microservices can make it hard to optimize. We will investigate the probabilistic model for microservice application performance. Our framework can potentially support more services, resources (e.g., network bandwidth allocation), and algorithms [6], [7], [18]; we plan to explore them in future work.

References

[1] Cloud adoption in 2020 – O'Reilly, 2020. Accessed: Dec. 2020. [Online]. Available: https://www.oreilly.com/radar/cloud-adoption-in-2020/

[2] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 3–18.

[3] K. Rzadca *et al.*, "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, Art. no. 16.

[4] H. Qiu *et al.*, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implement.*, 2020.

[5] Y. Gan *et al.*, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 19–33.

[6] H. Yuan, J. Bi, M. Zhou, Q. Liu, and A. C. Ammari, "Biobjective task scheduling for distributed green data centers," *IEEE Trans. Autom. Sci. Eng.*, to be published, doi: 10.1109/TASE.2019.2958979.

[7] Z. Lv, L. Wang, Z. Han, J. Zhao, and W. Wang, "Surrogate-assisted particle swarm optimization algorithm with pareto active learning for expensive multi-objective optimization," *IEEE/CAA J. Automatica Sinica*, vol. 6, no. 3, pp. 838–849, May 2019.

[8] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *CoRR*, vol. abs/1012.2599, 2010. [Online]. Available: http://arxiv.org/abs/1012.2599

[9] M. Pedram, "Energy-efficient datacenters," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 31, no. 10, pp. 1465–1484, Oct. 2012.

[10] J. D. McCalpin, "Stream benchmark," 1995. [Online]. Available: www.cs.virginia.edu/stream/ref.html# what

[11] D. Hernández-Lobato *et al.*, "Predictive entropy search for multi-objective Bayesian optimization," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1492–1501.

[12] C. K. Williams and C. E. Rasmussen, *Gaussian Processes for Machine Learning*. Cambridge, MA, USA: MIT Press, 2006.

[13] Kubernetes, 2020. Accessed: Dec. 2020. [Online]. Available: https://kubernetes.io/

[14] WRK, 2020. Accessed: Dec. 2020. [Online]. Available: https://github.com/wg/wrk

[15] Intel®RDT software package, 2019. Accessed: Dec. 2020. [Online]. Available: https://github.com/intel/intel-cmt-cat

[16] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[17] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 4292–4293.

[18] Q. Kang, X. Song, M. Zhou, and L. Li, "A collaborative resource allocation strategy for decomposition-based multi-objective evolutionary algorithms," *IEEE Trans. Syst. Man Cybern. Syst.*, vol. 49, no. 12, pp. 2416–2423, Dec. 2019.