Dynamic Multi-objective Scheduling of Microservices in the Cloud

Hamid Mohammadi Fard
Technical University of Darmstadt
Darmstadt, Germany
Email: fard@cs.tu-darmstadt.de

Radu Prodan
Klagenfurt Universität
Klagenfurt, Austria
Email: radu.prodan@itec.aau.at

Felix Wolf
Technical University of Darmstadt
Darmstadt, Germany
Email: wolf@cs.tu-darmstadt.de

***Abstract*—For many applications, a microservices architecture promises better performance and flexibility compared to a conventional monolithic architecture. In spite of the advantages of a microservices architecture, deploying microservices poses various challenges for service developers and providers alike. One of these challenges is the efficient placement of microservices on the cluster nodes. Improper allocation of microservices can quickly waste resource capacities and cause low system throughput. In the last few years, new technologies in orchestration frameworks, such as the possibility of multiple schedulers for pods in Kubernetes, have improved scheduling solutions of microservices but using these technologies needs to involve both the service developer and the service provider in the behavior analysis of workloads. Using memory and CPU requests specified in the service manifest, we propose a general microservices scheduling mechanism that can operate efficiently in private clusters or enterprise clouds. We model the scheduling problem as a complex variant of the knapsack problem and solve it using a multi-objective optimization approach. Our experiments show that the proposed mechanism is highly scalable and simultaneously increases utilization of both memory and CPU, which in turn leads to better throughput when compared to the state-of-the-art.**

***Keywords*-scheduling microservices; cloud computing; multi-objective optimization; knapsack problem; resource management;**

I. Introduction

Early monolithic cloud applications, with a rapid growth rate, are being replaced by microservice-based applications [1]. Nowadays, almost all cloud providers offer various microservice deployment services, as part of their fundamental services.

Microservices architecture is a progression of service-oriented architecture. In this approach, an application is decomposed as a collection of loosely coupled services, which interact over remote interfaces [2]. Microservices are tightly dependent on elasticity feature of cloud computing. Microservices architecture facilitates autoscaling of applications and considerably decreases cost of outsourcing of services to public commercial clouds. This is particularly important for IoT applications on the edge, as discussed in [3].

Ever-increasing number of microservices and critical need of realtime execution of these services highlights the problem of microservices scheduling for both cloud providers and service developers [4]. Using microservice calls, in public clouds, the customers do not pay by an hourly-based, pay-as-you-go pricing model but they pay only based on the number of service requests and duration of running services [5]. Therefore if a service provider can not perfectly utilize its available resources and provide the best possible response time to the service calls, all service customer, consumer and provider would be negatively impacted.

It is quite common to specify memory and CPU requests, in the service manifest [1], in most microservice deployment frameworks, such as Kubernetes [6]. The ratio between requested memory and CPU, in various services, could be completely uncorrelated. Then unilateral attempt to utilize a single resource (memory or CPU) could cause wasting the other resource of the cluster nodes. Moreover, we should notice that for deployment of microservices, resources of individual nodes can not be aggregated as integrated resources of cluster. Consequently, resource requirements of services and resource utilization of nodes should not be considered independently and we need to consider the impact of each component on the whole system.

In this paper, using memory and CPU requests specified for service deployment, we propose a novel scheduling mechanism that balances memory and CPU utilization of nodes and increases the utilization of the whole cluster, which in turn leads to better throughput. Our approach is simple, fast and very well scalable. The proposed mechanism is a general approach that can be used by developers for orchestrating their containers-based microservices (for instance, using Amazon Elastic Container Service) or can be used by enterprise cloud providers for orchestration of serverless microservices (for instance, AWS Lambda). Moreover, our approach is not bounded only to memory and CPU and could be applied for other resources (e.g. GPU), if the orchestrator supports request and limit specification for those resources in the service manifest.

In our problem, we focus on two critical objectives: utilization and throughput of cluster. We transform this problem, to sequence of a complex variant of the knapsack problem [7] and solve it using a multi-objective optimization approach. Considering the usual short lifetime and large

[1] https://kubernetes.io/docs/tasks/configure-pod-container/

scale of microservices in enterprise public clouds, low latency is a critical need of such scheduling mechanism, which has been carefully considered in our approach.

Our contribution in this paper is multifold. First, we proposed a general knapsack problem model that can be extended for various resource requests specified in the service manifest. Second, we considered simultaneous utilization of memory and CPU separately for each node and cumulatively in the whole cluster. Third, we used least knowledge, in advance, to dynamically schedule microservices that makes our approach robust in dynamic production environments.

The rest of the paper is organized as follows. In Section II, we review the related work and discuss how our approach is different. We model the problem formally in Section III. In Section IV, we propose a novel scheduling mechanism that is evaluated in Section V. Finally, we conclude the paper in Section VI.

II. Related Work

Scheduling of cloud resources in low level services, such as IaaS, has been extensively studied, in the last decade, e.g. [8] but less efforts have been devoted to high level services, such as microservice deployment. Although, almost the main goal of providers, in all schedulers, is to provide the best user experience according to the service level agreement (SLA) but resource utilization is always the objective of high priority for most providers. Scheduling mechanisms for different services confront with different challenges. In IaaS, the providers aim to allocate virtual machines to physical hosts while in microservice deployment, the providers need to allocate software services to virtual/physical nodes.

There are several popular frameworks for service orchestration, such as Docker Swarm [2], Mesos [3] and Rancher [4] and also there are cloud-specific technologies like Amazon elastic container service (ECS) [5] and Oracle application container cloud service [6] but Kuberenets is the so-called standard orchestration framework. Kubernetes is governed by cloud native computing foundation (CNCF) since 2015 and many of available frameworks have been produced based on that.

Microservices scheduling in orchestration frameworks directly impacts on the utilization of resources [9]. Improper scheduling mechanisms of orchestration frameworks can waste the resources quickly. On the other side, we could not spend long time to find the optimal solutions for short-life services. Subsequently, scheduling latency is a vital factor in such schedulers and scheduling overhead can not be simply neglected. In the rest of this section, we study some of valuable researches in domain of microservices scheduling.

[2]https://docs.docker.com/engine/swarm/

[3]http://mesos.apache.org/

[4]https://rancher.com/

[5]https://aws.amazon.com/ecs/

[6]https://docs.oracle.com/en/cloud/paas/app-container-cloud/index.html

A multi-objective scheduling approach in a cloud federation, proposed by [10], schedules microservices based on multiple user objectives, such as latency and cost. The final goal is to reduce the end-to-end latency and data communication between services of dependent service chains.

There are several network-aware scheduling approaches in related work, such as [11] and [3]. Network-aware approaches are particularly useful for running IoT services on the edge. In such environments, the services and resources are extensively distributed and considering the network topology in scheduling decision can prepare large potential for performance improvement of the system.

The authors of [12] propose a mathematical scheduling model and multiple classical approaches for scheduling of microservices in cloud-edge environments. The main focus of this work is using less general devices for running microservices.

A meta-scheduler has been proposed in [13] to fairly schedule the workloads of different users. In this approach, the scheduler calculates the overall resource demands and current resource consumption of each user and then considering the balance in resource consumption of users makes the final decision.

In [14], using an approximate Markov Decision Process, the authors propose an energy efficient scheduling by temporarily deactivating optional components of microservices. They mainly focus on overloaded datacenters and compares their solution with VM consolidation approach.

To the best of our knowledge, the microservices scheduling mechanisms usually consider single dimensional nodes. In other words, they mostly do not consider separate resources (e.g. memory and CPU) of nodes, individually in the scheduling decision. We believe that such scheduling approaches could waste the resources and decrease utilization and throughput of the clusters. Compared to the related work, our dynamic approach schedules the microservices by simultaneously considering memory and CPU requirements of services and memory and CPU capacity of nodes.

III. System Model

In this section, we formally define platform, workload, and problem models.

A. Platform Model

We assume a cloud provider dedicates the cluster $N = \{n_1, \ldots, n_p\}$, as the set of p connected nodes for serving microservices deployment. Nodes could include baremetal, virtual machines or any hybrid of those. Each node $n_j \in N$ is bounded by the capacity vector $n_j = (MEM_j, CPU_j)$, which respectively defines the memory and CPU limitations of the node.

B. Workload Model

The cloud provider hosts microservice-based applications of different customers. Each application is a collection of microservices that are requested dynamically based on variant arrival distributions. The logics behind the service calls' distributions are defined by the autoscaling policies of applications and therefore are out of the scope of this research.

The microservice instances/calls constitute a workload queue S including m microservices such that $S = [s_1, \ldots, s_q]$. Each microservice instance (or simply, service) $s_i \in S$ is characterized by its specification vector $s_i = (mem_i, cpu_i)$. The tuple presents memory and CPU requests of the service, respectively. The runtime/lifetime of each service $s_i \in S$ on each node $n_j \in N$ is denoted by $runtime^i_j$.

Since there is no direct communication between services, each service has its own private datastore then the inter-service relations only define the microservices chains. Therefore, we do not need to model the inter-service communication of microservices in our workload model. In other words, whenever a service invokes another service, the invoked service is appended as a new entry to the end of the service queue S.

This workload model could generally cover a broad range of microservice-based applications hosted on cloud environments or private clusters, for instance typical web applications designed based on serverless microservices architecture.

C. Problem Model

Our research question is scheduling of the microservices instances in S to the nodes presented by N. We denote this scheduling by the mapping function $sched : S \longmapsto N$. The cloud provider aims to maximize the throughput of the system, which is reflected by the number of completed services in a time interval Δt. The scheduler algorithm must be perfectly scalable and practically usable in large scale and enterprise cloud environments.

We formulate our scheduling problem as sequence of a complex variant of well-known knapsack problem. The knapsack problem is a combinatorial problem where a number of objects, each associated with a value and a weight, must be packed into a knapsack of specific capacity, such that the value of the objects within the knapsack is maximized [15].

Our scheduling problem is a multi knapsack problem, because there are p nodes $n_j \in N$ that must be packed by microservices instances $s_i \in S$. As defined in the platform model, each node n_j is bounded by two compute resources, namely memory limitation MEM_j and CPU limitation CPU_j. Then each knapsack has a 2-dimensional capacity vector including memory and CPU capacities, which represents the maximum weight that the node can support. In other words, each knapsack must be packed by the services, with two weights: memory and CPU requests. For packing the knapsacks, we aim to maximize the profit or value of each allocation decision based on two objectives: the average memory utilization and the average CPU utilization of the cluster, in a time interval Δt, between the starting time of the system and the call time of the service. In the evaluation of our scheduler mechanism, in Section V, we will discuss that the proposed profit function reflects the throughput of the system, as requested. Consequently, we modeled the microservices scheduling problem in cloud as a "Bounded Multi-Dimensional Multi-Objective Multiple Knapsack Problem".

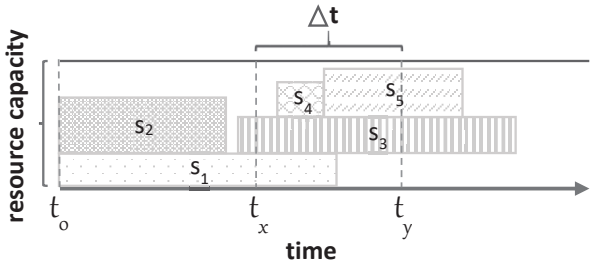


Figure 1: Resource utilization in the time interval Δt. To calculate the utilization of the resource in Δt, we need to consider the complete runtime for s_4 and the partial runtime for s_1, s_3 and s_5, which are bounded in $[t_x, t_y]$.

To formulate the capacity vector of each node, we need to distinctly model the utilization of memory and CPU. Utilization of these two compute resources in a time interval Δt are calculated by the following equations.

$$mem\_util(n_j, \Delta t) = \frac{\sum_{\{\forall\ s_i \mapsto n_j\}} \left(mem_i \cdot part(runtime^i_j, \Delta t) \right)}{MEM_j \cdot \Delta t} \tag{1}$$

$$cpu\_util(n_j, \Delta t) = \frac{\sum_{\{\forall\ s_i \mapsto n_j\}} \left(cpu_i \cdot part(runtime^i_j, \Delta t) \right)}{CPU_j \cdot \Delta t} \tag{2}$$

In these equations, $part(runtime^i_j, \Delta t)$ returns the part of runtime of the service s_i on the node n_j that is placed inside Δt. To better understand of the concept, a sample schedule histogram is represented in Figure 1,

Using equations 1 and 2, we define the average utilization of cluster. Because of different resource capacity of nodes, for average utilization of the cluster, we need to calculate the weighted average memory and CPU utilization of cluster, as denoted in the equations 3 and 4. To calculate the resources utilization, we ignored the idle nodes of the cluster, which have no service running at that time. In Section V, we

discuss how this assumption improves the schedule results.

$$\overline{mem\_util(\Delta t)} = \frac{\sum_{j=1}^{p} MEM_j \cdot mem\_util(n_j,\ \Delta t)}{\sum_{j=1}^{p} MEM_j} \quad (3)$$

$$\overline{cpu\_util(\Delta t)} = \frac{\sum_{j=1}^{p} CPU_j \cdot cpu\_util(n_j,\ \Delta t)}{\sum_{j=1}^{p} CPU_j} \quad (4)$$

The execution time of the service s_i allocated to the node n_j is defined as the time difference between submission time of the service (call time) and its completion time. The following equation calculates this time by sum of scheduling latency, waiting time for the node and runtime of the service.

$$exec\_time^i_j = shced\_latency^i + wait\_time^i_j + runtime^i_j \quad (5)$$

As discussed, the profit function for making each allocation decision is defined by a bi-objective vector, as follows. In Equation 6, Δt is the time interval between the starting time of the system and the call time of the service.

$$profit(s_i,\ n_j) = (\overline{mem\_util(\Delta t)},\ \overline{cpu\_util(\Delta t)}) \quad (6)$$

Finally, the problem of microservices scheduling in cloud is transferred to a sequence of bi-objective optimization problems, as follows:

$$\forall\ s_i \in S \wedge \forall\ n_j \in N\ \ maximize\ \ profit(s_i, n_j)$$

$$subject\ to\ \forall\ time\ t \wedge\ \forall\ n_j \in N \begin{cases} \sum_{\{\forall\ s_i \mapsto n_j\ \ in\ \ t\}} mem_i \leq MEM_j \\ \sum_{\{\forall\ s_i \mapsto n_j\ \ in\ \ t\}} cpu_i \leq CPU_j \end{cases}$$

The system model proposed in this section can be generalized and extended for other resources, such as local ephemeral storage of containers, accelerator resources, etc. provided that the orchestration framework supports the resource request specification in the service manifest.

IV. Least Waste, Fast First Algorithm

In Section III, we observed that our model is a complex variant of the knapsack problem. The knapsack problem is NP-complete problem, which has no exact solution in a polynomial time complexity [16]. Considering our extra constraints and objectives (see the subsection III-C), our problem is even more complex.

The scheduling of microservices in large scale is a time-critical problem that needs fast and efficient allocation decision. To solve the problem, we propose a greedy approach called **Least Waste, Fast First** (**LWFF**) in Algorithm 1.

First, for choosing the microservices from the service queue S, we rely on first come first served (FCFS) approach.

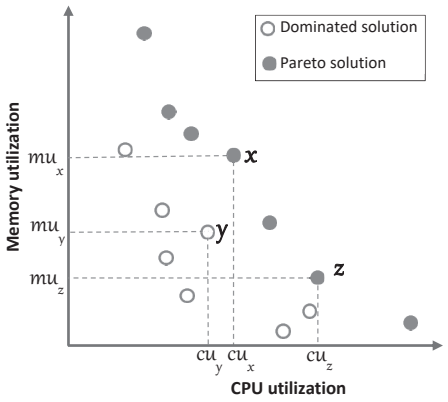


Figure 2: Concept of Pareto solutions. Point x dominates point y ($x \succ y$), while $mu_x > mu_y$ and $cu_x > cu_y$. Two points x and z are non-dominated solutions ($x \not\succ z$ and $z \not\succ x$) because $mu_x > mu_z$ and $cu_x < cu_z$.

Since usually the service queue is a long queue including tons of microservices, FCFS policy does not load extra overhead to the scheduling latency. To select the most adequate node for allocating a service, three phases are followed: (1) filtering, (2) producing the *Pareto* set and (3) choosing the final solution.

In line 7 of the algorithm, the set $feasible\_nodes$ is created that includes all nodes meeting the service requirements, such as affinity constraints ($feasible\_nodes \subset N$). In continue, the algorithm runs a fast multi-objective comparison based on the profit equation (see Equation 6), for all $n_j \in feasible\_nodes$.

In lines 8–12, to allocate a service s_i to each node $n_j \in feasible\_nodes$, we calculate the utilization of the whole cluster (see Equations 3 and 4), assuming the service is assigned to that node. We calculate the profit vector of each decision in line 10. When all profits are calculated for all possible schedule solutions, we remove the dominated solutions from the solution space (lines 13–20).

A schedule solution $s_i \mapsto n_x$ dominates another solution $s_i \mapsto n_y$ and is denoted by $s_i \mapsto n_x \succ s_i \mapsto n_y$, if the profit vector of $s_i \mapsto n_y$ has lower utilization, in both memory and CPU. If a solution $s_i \mapsto n_x$ can not dominate another solution $s_i \mapsto n_y$ we show it by $s_i \mapsto n_x \not\succ s_i \mapsto n_y$. As shown in Figure 2, the non-dominated solutions constitute a set called Pareto set.

In the next step, in lines 22–26, from the Pareto set (non-dominated solutions) generated in the previous step, we choose the solution that has the least execution time and finally, in line 27, the service is assigned to the chosen host.

V. Experimental Evaluation

We compared our proposed scheduling with two popular miscroservices scheduling approaches and also discuss the quality of LWFF solutions by comparison with the Pareto

Algorithm 1: Least Waste, Fast First (LWFF).

Input: Set of microservices instances: S; Set of cluster nodes: N
Output: Schedule decision: $sched : S \mapsto N$

begin
  $start\_time \leftarrow get\_time()$ /* set the starting point */
  while $S \neq \emptyset$ **do**
    $s_i \leftarrow deqeue(S)$ /* dequeue the head of S */
    $current\_time \leftarrow get\_time()$ /* get the wall-clock time */
    $profits \leftarrow \emptyset$ /* store the profit vectors */
    $feasible\_nodes \leftarrow \{n_j \in N \mid n_j \; meets \; s_i \; requirements\}$ /* filtering phase */
    foreach $n_j \in feasible\_nodes$ **do**
      $\Delta t \leftarrow [start\_time,\ current\_time + wait\_time^i_j + runtime^i_j]$ /* make the time interval */
      $profit(s_i,\ n_j) \leftarrow (\overline{mem\_util(\Delta t)},\ \overline{cpu\_util(\Delta t)})$ /* calculate the weighted average of memory and CPU utilization */
      $profits.append(profit(s_i,\ n_j))$ /* append the profit of $s_i \mapsto n_j$ to the profits list */
    end
    foreach $profit(s_i,\ n_j) \in profits$ **do**
      foreach $profit(s_i,\ n_k) \in profits$ **do**
        if $j \neq k \wedge profit(s_i,\ n_k) \succ profit(s_i,\ n_j)$ **then**
          $profits.remove(profit(s_i,\ n_j))$ /* remove dominated solutions */
          break
        end
      end
    end
    $host \leftarrow x \; such \; that \; profit(s_i,\ n_x) = head(profits)$ /* initiate *host* with the node of first element in *profits* list */
    foreach $profit(s_i,\ n_j) \in profits$ **do**
      /* find the node with shortest execution time */
      if $exec\_time^i_j < exec\_time^i_{host}$ **then**
        $host \leftarrow j$
      end
    end
    $sched.append(s_i \mapsto n_{host})$
  end
end

solutions achieved by the state-of-the-art in multi-objective optimization. As follows, we first explain the experimental setup and then we present and analyze the results.

A. Experimental setup

To evaluate LWFF mechanism, we ran an extensive set of experiments, using an extension of the *order management* application available on GitHub [7]. We developed LWFF algorithm as a web server and tested its scalability using JMeter [8]. We used jMetal [17] library to produce the Pareto set of schedule solutions by SPEA2 [18], which is a well-known multi-objective evolutionary algorithms. Inspiring by the instance types of AWS EC2 [9], we generated different instance templates with various resource flavors, as shown in Table I.

To have a various combination of problem size, we ran the experiments in 9 different classes, as depicted in Table II. This helped us to study the behavior of our scheduler regarding variant number of service calls and cluster sizes.

[7]https://github.com/PacktPublishing/Hands-on-Microservices-with-Python

[8]https://jmeter.apache.org/

[9]https://aws.amazon.com/ec2/instance-types/

Table I: Six node-templates used in the experiments.

type name	memory (GB)	vCPU
t3.nano	0.5	2
t2.micro	1	1
a1.medium	2	1
c4.large	3.75	2
c5n.large	5.25	2
m4.xlarge	16	4

Table II: Various problem sizes used in the experiments.

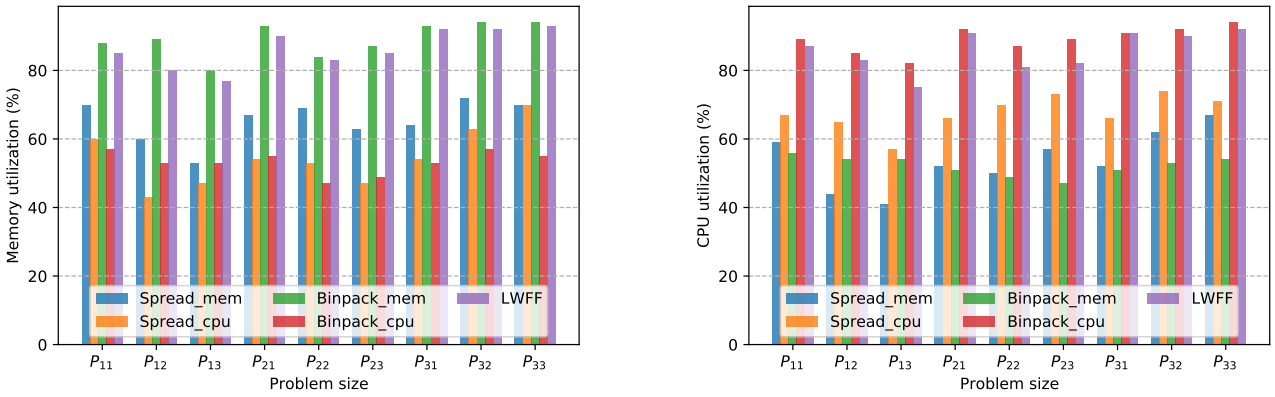
		nodes no.		
		10	50	200
services no. per seconds	100	P_{11}	P_{12}	P_{13}
	1000	P_{21}	P_{22}	P_{23}
	10000	P_{31}	P_{32}	P_{33}

B. Experimental results

In the first part of the experiments, we compared LWFF with the two common microservices scheduling mechanisms: *Spread* and *Binpack*. Spread strategy balances the cluster nodes by selecting the nodes with the least load and the goal of Binpack is to maximize the utilization of nodes. To have a more precise comparison, we implemented two variants of Spread and Binpack scheduling strategies. *Spread_mem* and *Spread_cpu* aim to balance memory and CPU utilization, respectively among all nodes of the cluster. Similarly, two variants of Binpack, namely *Binpack_mem* and *Binpack_cpu* target to fully utilize nodes considering memory and CPU capacities, independently.

Figure 3 compares the average utilization of memory and CPU in the cluster, while using Spread, Binpack and LWFF schedulers. To calculate the average of resource utilization, as discussed in subsection III-C, we have considered only active nodes (against the idle nodes), who host at least one microservice running. As shown in Figure 3a, utilization of Spread_cpu and Binpack_cpu are lower than the other algorithms, because both algorithms make the schedule decision based on CPU utilization and ignore memory utilization. The same trend can be viewed in Figure 3b for Binpack_mem and Spread_mem. Although Binpack_mem in Figure 3a and Binpack_cpu in Figure 3b provide the best memory and CPU utilization independently but considering both memory utilization and CPU utilization simultaneously, LWFF outperforms all competitors. We should notice that in low traffic scenarios (e.g. P_{12} and P_{13}), the utilization of cluster is generally low for all scheduling strategies.

We measured the scheduling latency and its impact on the execution time of services, as depicted in Figure 4. Figure 4a shows the average latency to achieve the schedule solutions. We could observe that all competitors have lower scheduling latency than LWFF but in Figure 4b, we could



(a) The average of memory utilization.

(b) The average of CPU utilization.

Figure 3: The comparison of resource utilization of nodes. For better perception of schedule outcome, the utilization has depicted in separate charts for memory and CPU.

view that the services running by LWFF have considerably lower execution time. Considering the fact that execution time is sum of scheduling latency, waiting time and runtime, clarifies that the schedule solutions by LWFF have lower waiting time and runtime, which cover higher scheduling latency of LWFF.

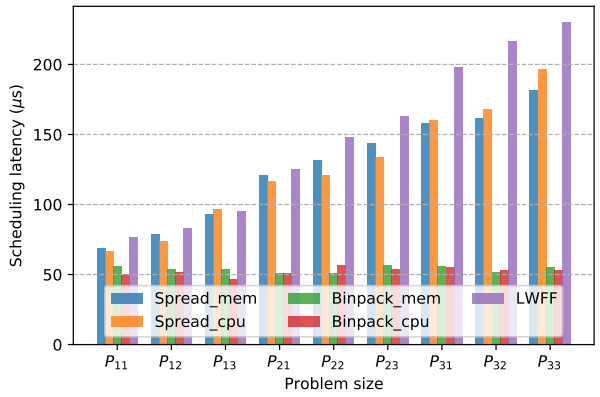
In the next part of the experiments, we measured the throughput of the cluster. Considering the previous experiments, while LWFF provides better resource utilization and execution time, then we may implicitly conclude that the throughput of LWFF is better too. We analyzed this claim experimentally. In Figure 5, we depicted the average of throughput for active nodes per second. As shown in Figure 5, the throughput of LWFF is remarkably higher than the other algorithms. These results strengthen our previous results shown in Figure 4b. Higher throughput of LWFF solutions in Figure 5 and less execution time of services shown in Figure 4b confirm that the profit function, as defined in Equation 6, was a proper choice.

The next experiment represents how LWFF tends to fully utilize the cluster nodes. This is particularly important when we are using virtual clusters on public clouds, because lower number of active nodes means less need to launch new nodes and consequently less monetary cost. In a private in-house cluster this could also help to consolidate resources and consume less energy [19]. In Table III, both variants of Spread have no idle nodes because they tend to use all nodes to keep the load balance among the cluster. The main logic behind Binpack is decreasing the number of active nodes by fully utilize nodes singly. Then we may expect that the lowest number of idle nodes between Spread, Binpack and LWFF belongs to Binpack scheduling. But considering simultaneous memory and CPU utilization, LWFF outperforms both Binpack_mem and Binpack_cpu, dramatically.

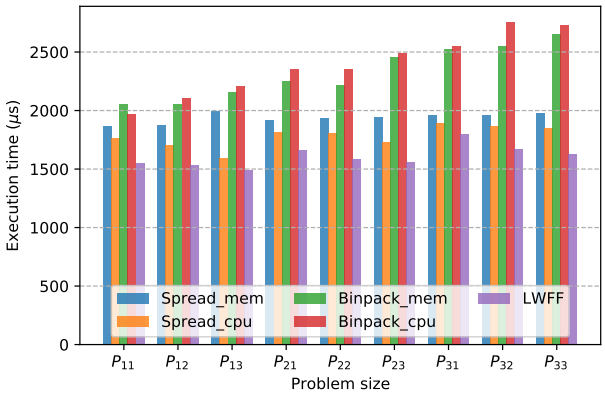
In continue, we considered the *coverage* metric [20] to compare LWFF with SPEA2. The coverage of a set A on a set B indicates how many members of B are dominated by the members of A. The precondition to approximate the Pareto set using SPEA2 in our problem is having static services in the service queue S. To mimic this condition and make SPEA2 comparable with LWFF in our dynamic environment, we assumed three time intervals $\Delta t = 10$, $\Delta t = 30$ and $\Delta t = 60$ such that the number of services have no change in these intervals. Moreover, we configured SPEA2 to generate 5 Pareto solutions. We assumed two objectives for the schedule solutions: schedule latency and throughput. We observed that no Pareto solution estimated by SPEA2 was able to dominate any of the solutions provided by LWFF. In other words, the coverage of SPEA2 on LWFF in all scenarios is zero. The main reason behind this is high time complexity of SPEA2 to approximate a Pareto set. Although in our experiments, we observed that some of the Pareto solutions estimated by SPEA2 have relatively lower runtime than the solutions provide by LWFF but considering the high schedule latency for approximating the Pareto set in addition to lack of supporting dynamic microservice environments make such Pareto set approximation approaches hardly acceptable in the problem of microservice scheduling. On the other side, as shown in Figure 6, the coverage of LWFF on SPEA2 is remarkably high and also the coverage is increasing by growing the problem size. Consequently, LWFF was able to provide high quality schedule solutions in a reasonable time latency.

VI. Conclusion

In this paper, based on the knapsack problem, we proposed a general model for scheduling of microservices in



(a) The average of scheduling latency.



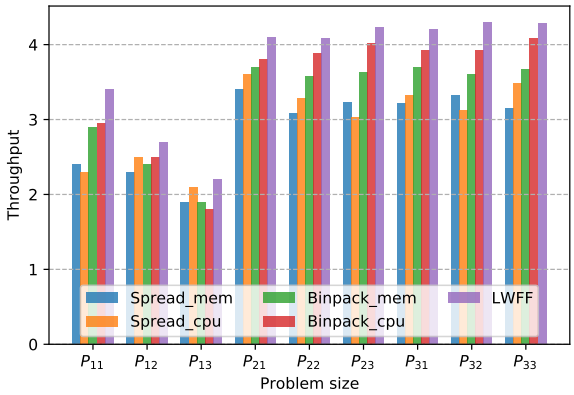
(b) The average of execution time.

Figure 4: Scheduling latency and its impact on the execution time of services. Scheduling latency is an important factor to calculate the execution time of microservices, particularly for short life services. Nonetheless, there is no direct correlation between scheduling latency and execution time.

Table III: The number of idle nodes in the cluster. Whatever this number is higher, we have some unused nodes that can we turn them off or release them. This means saving money and energy. The data has have been measured in three time intervals with the length of 10, 30 and 60 seconds, starting from the begin of experiments.

	$\Delta t = 10s$									$\Delta t = 30s$									$\Delta t = 60s$								
	P_{11}	P_{12}	P_{13}	P_{21}	P_{22}	P_{23}	P_{31}	P_{32}	P_{33}	P_{11}	P_{12}	P_{13}	P_{21}	P_{22}	P_{23}	P_{31}	P_{32}	P_{33}	P_{11}	P_{12}	P_{13}	P_{21}	P_{22}	P_{23}	P_{31}	P_{32}	P_{33}
Spread_mem	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Spread_cpu	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Binpack_mem	2	23	79	0	0	15	0	0	0	0	2	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Binpack_cpu	2	27	83	0	0	17	0	0	0	0	4	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LWFF	3	31	94	0	0	20	0	0	0	0	7	18	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0

Figure 5: The average of throughput of active nodes per second. To do a fair comparison we considered only active nodes and ignored idle nodes of the cluster.



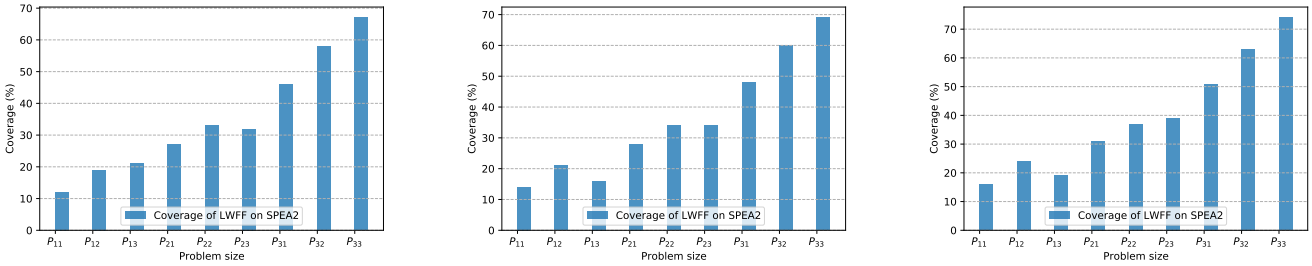
the cloud. This model can be extended for various resource requests specified in the service manifest of an orchestration framework. We solved the problem using a multi-objective optimization approach. Measuring the throughput, we experimentally presented that our proposed scheduler outperforms Spread and Binpack scheduling mechanisms. Moreover, using the coverage metric, we showed that our proposed approach provides efficient solutions compared to the approximated Pareto set generated by SPEA2. As a next step, we intend to use our scheduling mechanism for autscaling of microservices in the cloud.

Acknowledgement

This research has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 785907 (Human Brain Project SGA2) and the Specific Grant Agreement No. 945539 (Human Brain Project SGA3).

References

[1] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, p. 29–45, Jan. 2018.



(a) $\Delta t = 10$. (b) $\Delta t = 30$. (c) $\Delta t = 60$.

Figure 6: The coverage of LWFF on SPEA2. The coverage is the percentage of schedule solutions of LWFF, dominating the solutions estimated by SPEA2. Notice that the coverage of SPEA2 on LWFF is zero, in all problem sizes.

[2] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016, pp. 44–51.

[3] H. M. Fard, R. Prodan, and F. Wolf, "A container-driven approach for resource provisioning in edge-fog cloud," in *ALGOCLOUD 2019. Lecture Notes in Computer Science (LNCS), vol 12041. Springer, Cham.*, 2020, pp. 59–76.

[4] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[5] P. Rosati, F. Fowley, C. Pahl, D. Taibi, and T. Lynn, "Right scaling for right pricing: A case study on total cost of ownership measurement for cloud migration," *Cloud Computing and Services Science*, p. 190–214, 2019.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, p. 50–57, Apr. 2016.

[7] S. Martello, "Knapsack problems: Algorithms and computer implementations," *Wiley-Interscience series in discrete mathematics and optimization*, 1990.

[8] H. M. Fard, S. Ristov, and R. Prodan, "Handling the uncertainty in resource performance for executing workflow applications in clouds," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, 2016, pp. 89–98.

[9] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 159–169.

[10] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Multi-objective scheduling of micro-services for optimal service function chains," in *2017 IEEE International Conference on Communications (ICC)*, 2017, pp. 1–6.

[11] J. Santos, T. Wauters, B. Volckaert, and F. D. Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 351–359.

[12] I. Filip, F. Pop, C. Serbanescu, and C. Choi, "Microservices scheduling model over heterogeneous cloud-edge environments as support for iot applications," *IEEE Internet of Things Journal*, vol. 5, no. 4, pp. 2672–2681, 2018.

[13] A. Beltre, P. Saha, and M. Govindaraju, "Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters," in *2019 IEEE Cloud Summit*, 2019, pp. 14–20.

[14] M. Xu and R. Buyya, "Energy efficient scheduling of application components via brownout and approximate markov decision process," in *Service-Oriented Computing*. Cham: Springer International Publishing, 2017, pp. 206–220.

[15] N. Kumaraguruparan, H. Sivaramakrishnan, and S. S. Sapatnekar, "Residential task scheduling under dynamic pricing using the multiple knapsack method," in *2012 IEEE PES Innovative Smart Grid Technologies (ISGT)*, 2012, pp. 1–6.

[16] D. Pisinger, "Where are the hard knapsack problems?" *Computers & Operations Research*, vol. 32, no. 9, pp. 2271 – 2284, 2005.

[17] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760–771, 2011.

[18] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, pp. 95–100, 2001.

[19] A. Corradi, M. Fanelli, and L. Foschini, "Vm consolidation: A real case based on openstack cloud," *Future Generation Computer Systems*, vol. 32, pp. 118 – 127, 2014.

[20] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A comparative case study and the sstrength pareto approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.