Published in IEEE Systems Journal (Volume: 18, Issue: 2, June 2024)

Efficient and Flexible Component Placement for Serverless Computing

Shouxi Luo, *Member, IEEE*, Ke Li, Huanlai Xing, *Member, IEEE*, and Pingzhi Fan, *Fellow, IEEE*

***Abstract*—Nowadays, serverless computing has been widely employed and viewed as the new paradigm of cloud computing. Technically, serverless applications are made up of function components, which are packaged as specific layered files named container images. In production, different components are designed to partially share layers; and during the deployment, the hosting servers have to download the missing layers first, which might dominate the application startup delay.**

In this paper, we look into optimizing the deployment of serverless applications under the operational goals of *energy saving* and *load balance*, by exploring the reusability among involved container images to conduct content-aware component placements explicitly. We find that the two involved optimization problems can be formulated as *multi-objective (mixed) integer linear programs*, and prove that their common building block of minimizing the weighted sum of deployment cost for a given set of serverless components is NP-hard. To be practical, we develop an efficient yet flexible heuristic solution named BFGP (Best Fit Greedy Placement) which involves three variants BFGP-Full, BFGP-ES, and BFGP-LB for the problem. Performance studies show that BFGP is effective, expressive, and efficient. It not only achieves near-optimal placement very efficiently but also supports high-level operational policies like *energy saving* and *load balance*.

***Index Terms*—Cloud, service placement, serverless, energy saving, load balance**

I. Introduction

To simplify the use of cloud resources while providing a more fine-grained cost model for elastic applications, cloud providers like Amazon, Google, and Microsoft recently propose the new service model of Function as a Service (FaaS), making the wave of serverless computing [1, 2]. In emerging serverless computing, the implementation of an application is decoupled into multiple stateless, event-driven components (i.e., functions). During the processing, these function components are launched on-demand and in pipeline respecting the dynamic workload. Instead of peer-to-peer messaging, they communicate with each other using high-performance back-end storage and message services provided by the cloud provider (e.g., AWS S3, Amazon SQS). Such a paradigm makes the deployed applications super easy to scale from zero to "infinity", and greatly simplifies the operation needed by developers, thus becoming more and more popular. Indeed, serverless computing is predicted as the dominant cloud computing paradigm in the near future [1], and has already been widely employed for applications like web service, data processing, internet-of-things, machine learning, etc [3, 4]. Accordingly, optimizing the performance of serverless platforms has become a crucial objective for cloud providers [2].

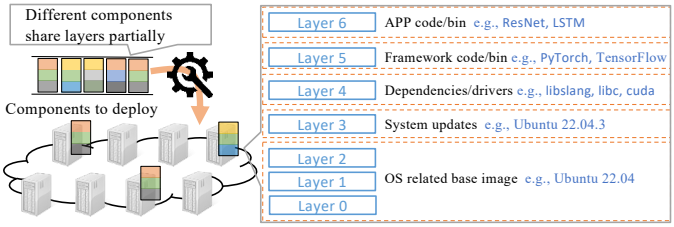


Fig. 1: An example of deploying containerized components for serverless applications in clouds.

Nowadays, serverless computing platforms like AWS Lambda, Azure Function, IBM OpenWhisk, and Kubeless, typically use *Container*, a lightweight virtualization technique, to manage their function instances [5, 6]. According to the workflow of how a containerized function component comes online, the lifecycle of a serverless function, on either a physical or virtual machine, can be split into two phases as Figure 1 shows: 1) fetch the involved component's container image(s) to *deploy*, then 2) instantiate the function instance(s) on-demand to *run*, which together dominate the startup delay of serverless application. In production, these two processes are triggered in multiple scenarios including fresh deployment, service migration, and scaling-up. As a shorter startup time yields a lot of advantages to both the system and application, optimizing the deployment delay is important and urgent [7].

Recently, a lot of efforts have been put into the aforementioned optimization [8–14]. Basically, these proposals mainly focus on designing schemes to mask the deploy time by pipelining the delivery of container images and cold-start of function instances [7, 15], or to accelerate the transfers of container images in deployment using advanced transport protocols [11, 12] or peer-to-peer system [13, 14]. Unfortunately, these solutions overlook the fact that the choice of the hosting servers for application components essentially has significant impacts on the deployment, as it determines

Manuscript received 2 July 2023; revised 12 January 2024; accepted 14 March 2024. The work of Shouxi Luo was supported by NSFC Project No.62002300; the work of Ke Li was supported by NSFC Project No.62202392, Project No.NDS2022-1 of Network and Data Security Key Laboratory in Sichuan Province, and NSFSC Project No.2023NSFSC0459; the work of Pingzhi Fan was supported by NSFC Project No.U23A20274. (Corresponding author: Shouxi Luo.)

Shouxi Luo, Ke Li, and Huanlai Xing are with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu, 611756, China, and with the Engineering Research Center of Sustainable Urban Intelligent Transportation, Ministry of Education, China (e-mail: sxluo@swjtu.edu.cn, keli@swjtu.edu.cn, hxx@swjtu.edu.cn).

Pingzhi Fan is with the Key Laboratory of Information Coding and Transmission, CSNMT Int Coop. Res. Centre, Southwest Jiaotong University, Chengdu, 611756, China (e-mail: p.fan@ieee.org).

the traffic volumes that the fetch of container images would trigger, resulting in performance loss. More specifically, as Figure 1 shows, in practice, a container image is made up of multiple layers; different functions' images are likely to share layers partially and a server generally hosts multiple function instances by design. Hence, by carefully co-locating function instances with shared layers on the same server, it is possible to reduce the amount of data to transmit during the deployment. Indeed, as we will show in §II, a well-designed service deployment scheme not only reduces the startup delay, but also leads to smaller storage- and network- footprints, and alleviates the impacts of service deployment on other co-located running applications. Thus, serverless platforms desire performance-optimized service deployment schemes. Although recent works [16] and [17] try to explore the layered and shareable structures of docker images to speed up the startup of service, they ignore the high-level operational goals like *energy saving* and *load balance* that the system might also pursue, thus are far from optimal for these operational goals.

To fill the gap, in this paper, we look into the problem of how to optimize the deployment of serverless applications respecting the operational goals of *energy saving* and *load balance*, with content-aware component placement. Indeed, besides FaaS-based serverless computing, our proposed schemes can be applied to other container-based platforms like Platform as a Service (PaaS) and Container as a Service (CaaS) as well.

Specifically, we build mathematical models to explicitly formulate the problem of cost-efficient placement of containerized serverless applications under these two operational goals and find that they are NP-hard in theory. Accordingly, we design BFGP (Best Fit Greedy Placement), an efficient yet flexible heuristic algorithm involving the variants of BFGP-Full (for purely deployment cost minimization), BFGP-ES (for deployment cost optimization & energy saving), and BFGP-LB (for deployment cost optimization & load balance) to conduct practical placements. The key design insights behind BFGP is that, by placing components on servers one-by-one and limiting the candidate set for each round of placement appropriately, we can pursue the scheduling goal of either *energy saving* or *load balance* with the same scheduling framework. Simulation results based on synthesized traces confirmed that BFGP achieves near-optimal performances in terms of the involved traffic volume, and is expressive to implement high-level operational policies like *energy saving* and *load balance*. Hence, it is able to reduce both the startup delay and network interference in co-located running services.

The main contributions of this paper are four-fold:

- A thorough analysis that identifies the benefits of content-aware placement of containerized components, along with the challenges for the design of a practical solution (§II).
- Math models that formulate the content-aware component placement optimization problem under the operational goals of *energy saving* and *load balance* explicitly for serverless application (§III).
- BFGP, an efficient and flexible algorithm framework, together with the variants of BFGP-Full, BFGP-ES, and BFGP-LB, to achieve effective placements of containerized components, respecting policies like *energy saving* and *load balance* (§IV).
- Extensive simulations that assess the effectiveness, expressiveness, and efficiency of the proposed BFGP (§V).

Finally, a short discussion of related works follows in §VI. Conclusions and possible future works are presented in §VII.

II. Background and Motivation

A. Reusability Among Layered Container Images

In serverless computing, the implementation of a function component involved in service, along with all its dependencies, is usually packaged as a container image. Based on this image, container runtime platforms like Docker can create and launch instances to handle the dynamic workload on-demand [1]. As Figure 1 shows, in production, a container image is generally made up of multiple layers. According to the container's design, these images must remain self-contained; as a result, the images of different function components might rely on common dependency files such as the same libraries and software frameworks. For example, they may run on Ubuntu 22.04, require the same libraries such as libslang, libstdc++, libc, and GPU drivers, or use the same version of Java runtime virtual machine. Thus, different container images share layers, saving both the storage and network footprints.

Moreover, with the employment of emerging layer optimization techniques [18], we argue that the reusability of layers among container images would continue to increase; hence, content-aware containerized component deployment is attractive to modern cloud systems.

B. Benefits of Context-aware Component Placement

Specifically, when different container images share layers, by co-locating function components whose images share layers on the same server, we can obtain the following benefits.

- **Smaller storage and network footprints**: The direct advantage of content-aware placement is that, for each layer, the hosting server would just download the layer only once and hold one copy for it in the local storage, resulting in saved network and storage footprints.
- **Lower startup delays**: According to most of the current commercial designs, a container instance could not be launched until its involved container image is already. A smaller network footprint means that the time needed for "downloading" is reduced. Thus, content-aware placement would lead to smaller startup delays as well.
- **Less network interference**: In production, there are generally many other services coexisting in the same cluster, some of which might provide user-facing services, and thus are very sensitive to network delays introduced by burst traffic. A smaller network footprint also means reduced network impacts of image downloading over co-located active time-sensitive applications.

C. Limitations of Existing Schemes

Regarding the placement of containerized components for continually incoming requests, modern industrial-grade serverless computing platforms like OpenWhisk, AWS Lambda, and

Azure Functions, generally rely on simple greedy schemes—despite various advanced optimization designs having been developed [10, 19, 20], at their core, multiple deployment tasks are treated as being able to reuse images, if and only if they are triggered by the invocations of the same function, without taking advantage of the partial reusability of layers among different functions' container images. For example, the default scheduler of OpenWhisk places the duplicated requests of the same function to the same yet randomly selected server to reduce cold start in a load-agnostic way [19]; AWS Lambda employs a similar consolidation policy but would select a new server in case those servers already in use do not have enough remaining resources [19].

Recently, an increasing amount of effort has been put into conducting layer-aware placement optimization of containerized components and services in the context of mobile edge computing (MEC) [21–23]. However, there are several significant differences between these prior studies and our work, making these existing proposals unable to deal with the problem this paper aims to solve.

Firstly, in MEC, edge servers are geographically distributed and networked with bandwidth-limited and heterogeneous WAN connections. As a result, the communication cost between the related components (e.g., those belonging to the same application) and the final end-users (e.g., mobile phones) must be taken into account[21–23]. While in intra-datacenter serverless computing, the underlying physical servers are networked with ultra low-latency, high-bandwidth, non-blocking datacenter networking techniques (e.g., RoCEv2 [24]); different functional components of the same application generally communicate with each other using the high-performance back-end storage and message services provided by the cloud provider, and the locations of components have little impact on the communication cost to the users outside the cloud.

Secondly, besides reducing the cost of downloading container images, modern cloud providers might also pursue changeable high-level operational goals like *energy saving* and *load balance* [20, 25]. More specifically, to improve performance efficiency and accelerate the completion of tasks, the operators might prefer to distribute the workload to more machines. However, more active machines generally lead to larger energy consumption and thus higher overall costs. Thus, to enhance the pricing competitiveness for service providers and sustainable computing, the operational policies of *energy saving* might be preferred in some cases [26, 27]. Moreover, nowadays, an increasing number of cloud service providers are powered by renewable yet unstable energy such as solar, wind, and hydro; and once the renewable energy is insufficient, the stored or brown energy would be used [26, 28]. Accordingly, the price of energy is likely to be unstable, leading to the shift of operational policies from *load balance* to *energy saving*; and *vice versa*. Hence, we argue that supporting the operational goal of *energy saving* and *load balance* is attractive for deploying containerized components for modern intra-datacenter serverless computing.

D. Design Challenges

All the above observations motivate us to design efficient algorithms to conduct content-aware placement for containerized serverless applications. However, to be a practical solution, the proposed scheme should hold the following properties.

- **Effectiveness**: First of all, since content-aware application deployments could bring with us the aforementioned excellent benefits, to obtain these advantages as much as possible, the proposed algorithm should be effective to achieve optimized service deployment.
- **Expressiveness**: Secondly, large-scale data centers generally involve more than one type of computing and network hardware, each with a different capacity. For instance, a portion of the servers might get equipped with the newest or a specific version of GPU, FPGA, or TPU cards, which are required by some of the applications in turn. Also, in different time periods, operators might prefer diverse high-level policies like *consolidation* for energy saving, or *dispersion* for load balance. Accordingly, the proposed solution should be expressive to deal with heterogeneity of hardware and diversity of policy.
- **Efficiency**: Last but not least, the time cost of the decision process might introduce non-trivial delays to the deployment of cloud service, especially when the scale of both the underlying data center cluster and deployment task is huge. To be practical, the proposed algorithm must run fast enough to obtain optimized deployments for online requests within a reasonable time.

III. Problem Description

In this section, we build two multi-objective (mixed) integer linear program (ILP) models to precisely describe the problems of content-aware service deployments (§III-A) under the operational goals of *energy saving* and *load balance*, respectively. We find that they are expressive to support hardware heterogeneity (§III-B), and the problems are NP-hard in theory (§III-C). Table I summarizes all the used notations.

A. Model

Without loss of generality, let's consider the task of deploying n function components to a cluster consisting of m machines/servers. These containerized components might belong to the same application, or multiple applications in case batched service deployments are employed. For component i, we use S_i to denote the set of its available candidate hosting servers, d_i to denote its resource requirement and binary variable $x_{i,j}$ to indicate whether function component i would be deployed at server j, respectively. In case multiple instances are needed to launch for a component, we can simply duplicate the component and treat them as individuals in deployment. As each component must be hosted by a server, we have the following constraints:

$$\sum_{j \in S_i} x_{i,j} = 1, \quad \forall i. \tag{1}$$

For these m servers, suppose that some of them have already hosted applications and others have not thus are in the low

TABLE I: Table of notations

Term	Meaning
n	Number of function components
m	Number of machines/servers
l	Number of distinct layers
i, j, k	Indexes of function components, servers, and distinct layers
$a_{j,k}$	1 if layer k is hosted on server j already, or 0 otherwise
b_j	1 if server j is active already, or 0 otherwise
c_j	Remaining capacity of server j
$\bar{c}_j$	Total capacity of server j
d_i	Resource demand of component i
g_j	Energy cost of activating server j
$w_{j,k}$	Cost of delivering layer k to server j
L_i	The set of layers involved by component i
S_i	The set of servers that can host component i
$x_{i,j}$	0-1 variable, whether component i is hosted by server j
$y_{j,k}$	0-1 variable, whether layer k should be delivered to server j
z_j	0-1 variable, whether server j is active after the deployment
ρ	Real-valued variable, the maximum server load after deployment

power state for energy saving. We use b_j with the value of 1 or 0 to denote the current status of server j (i.e., whether it is active or not), and g_j to denote the additional energy cost if server j is active. We further assume that the remaining capacity of server j is c_j, and define the binary variable z_j to denote whether this server must be activated after the deployment. Then, we have:

$$b_j \leq z_j, \quad \forall j, \tag{2}$$

and the aggregated resource demand of its hosted components should not exceed this limit:

$$\sum_{i:j\in S_i} d_i x_{i,j} \leq c_j z_j, \quad \forall j. \tag{3}$$

We assume that there are l distinct image layers in total and the set of required layers for the i-th component is denoted by set L_i. On the server side, we use $a_{i,k}$, a constant binary with the value of 1 or 0, to indicate whether this server already holds layer k or not, and employ the 0-1 variable of $y_{j,k}$ to indicate whether server j needs to fetch layer k. In case the target server does not have some required image layers, it has to download them from the image registry, yielding the following constraints for $a_{j,k}$, $x_{i,j}$, and $y_{j,k}$.

$$x_{i,j} \leq a_{j,k} + y_{j,k}, \quad \forall (i,j,k). \tag{4}$$

Minimum deployment cost & energy saving. By taking all the above into account, it is straightforward to formulate the problem of finding the component placement with both the minimum fetching cost (e.g., time cost) for fast startup and the minimum number of active servers for energy saving, for the given set of serverless components as a multi-objective integer linear program of (6). Here, $w_{j,k}$ is a tunable parameter denoting the cost it takes for the server j to fetch layer k.

$$x_{i,j}, y_{j,k}, z_j \text{ are binary}, \quad \forall (i,j,k), \tag{5}$$

$$\text{Minimize} \left[\sum_{j=1}^{m} \sum_{k=1}^{l} w_{j,k} y_{j,k}, \sum_{j=1}^{m} g_j z_j \right] s.t. \{(1)-(5)\}. \tag{6}$$

Minimum deployment cost & load balance. Suppose that the original capacity of server j is $\bar{c}_j$. By using the variable of ρ to denote the maximum server load after the deployment, we have the constraints of (7). Accordingly, the problem of finding the optimal component placement yielding the minimum fetching cost (i.e., time cost) and balanced loads can be formulated as the multi-objective mixed-integer linear program of (8).

$$\rho \geq \frac{\bar{c}_j - c_j + \sum_{i:j\in S_i} d_i x_{i,j}}{\bar{c}_j}, \quad \forall j, \tag{7}$$

$$\text{Minimize} \left[\sum_{j=1}^{m} \sum_{k=1}^{l} w_{j,k} y_{j,k}, \rho \right] s.t. \{(1)-(5), (7)\}. \tag{8}$$

Note that the original and remaining capacity of server j are denoted by $\bar{c}_j$ and c_j, respectively; thus, $\frac{\bar{c}_j - c_j + \sum_{i:j\in S_i} d_i x_{i,j}}{\bar{c}_j}$ in the constraint (7) represents server j's normalized load after this round of complement deployments. Besides the fetching cost, i.e., $\sum_{j=1}^{m} \sum_{k=1}^{l} w_{j,k} y_{j,k}$, the model of (8) also tries to minimize the maximum load among all servers, i.e., ρ.

B. Expressiveness

Notably, for the i-th task, S_i is configured by the operator, implying the set of server nodes that this function component could be placed at. Likewise, the value of involved $w_{j,k}$ is configurable as well, denoting the normalized cost of downloading layer k to server j. In production, large-scale data centers often involve multiple types of hardware configurations, resulting in heterogeneous compute- and forwarding- capacities among server nodes. By selecting S_i, the set of candidate servers for each task i, properly, constraints (1) and (3) can precisely capture the specific hardware requirements involved by each component. In addition, let us denote the normalized throughput of fetching layer k with volume v_k to server j by $t_{j,k}$, and further define $w_{i,j}$ as $\frac{v_k}{t_{j,k}}$. Then, the heterogeneity of the bandwidth has been embodied in the objectives of (6) and (8) and a deployment with less fetching cost is preferred.

C. Hardness

It is obvious that the common building block of models (6) and (8) is to minimize the deployment costs for the given components with proper placements, which is an Integer Linear Program (ILP). As Theorem 1 clarifies, it is NP-hard.

Theorem 1. *The common building block of* (6) *and* (8)*, i.e., minimizing the weighted sum of deployment cost for the given components, is NP-hard.*

Proof. Indeed, determining the feasibility of the problem is NP-hard as well. We conduct the proof by reducing the well-known NP-complete *subset-sum problem* [29] to it. Consider that there are two homogeneous servers A and B, whose remaining capacities are c_A and c_B, respectively. Then, the task is to deploy n components whose resource demands are

$d_1, d_2, \cdots, d_n$, to these two servers, respectively. Without loss of generality, we assume that c_A, c_B, and $d_i (i = 1, \cdots, n)$ are all integers and their values satisfy: $\sum_{i=1}^{n} d_i = c_A + c_B$. In such a situation, to figure out whether there exists a feasible placement, we have to solve the NP-hard *subset-sum problem* of finding a subset of tasks summing to c_A. □

IV. Solutions

As (6) and (8) are NP-hard problems, we propose BFGP (Best Fit Greedy Placement), a heuristic yet expressive placement algorithm involving the variants of BFGP-Full, BFGP-ES, and BFGP-LB, based on the problem characteristics.

In production, industrial-grade container platforms like K8S, Docker Swarm, and Mesos [30], and serverless computing platforms like OpenWhisk and Azure Function [10, 19, 20] generally employ inner controllers for workload and resource management in the control plane. Thus, it is straightforward to deploy our proposed schemes in these controllers, and engineering efforts are needed to carry out the implementation.

A. Design Insights

Obviously, the main benefits of the content-aware placement come from the facts that: 1) given a function component if a server already hosts parts or all of the layers it involves, deploying the component on this server would be a good choice as there is no need to transmit these hosted layers; 2) likewise, in case two function components share many layers, then placing them on the same server would greatly reduce the deployment cost as well. Based on these observations, a reasonable heuristic design is to i) find the placement of function components one by one; and ii) for each component, place it on the server with the smallest cost greedily. In our design, we use *tuples* to denote the weights used for server comparison. Just like the *tuple* type in Python programming language, two tuple values are compared lexicographically using the comparison of their corresponding elements.

Regarding the objectives like *energy saving* and *load balance*, we find that by dynamically selecting the candidate server set of each component to encode such high-level requirements, we can straightforwardly achieve multi-objective complement placement. To do so, for the i-th component's candidate server j with the remaining capacity of c_j, we define its selection weight as p_j, which is a tuple value related to the values of d_i, c_j, and the type of targeted operation objectives (i.e., energy saving or load balance). Then, for the deployment of component i whose candidate servers for the deployment are S_i, we only consider the $\lfloor \kappa |S_i| \rfloor$-largest servers from S_i, where $0 < \kappa \leq 1$. Here, κ is a tunable parameter controlling the trade-offs BFGP would make between the two objectives involved in models (6) and (8). A larger κ would bring more benefits to the minimization of deployment cost: when $\lfloor \kappa |S_i| \rfloor = 1$, BFGP degenerates into the design that pursues the single goal of either *energy saving* or *load balance*; and once $\kappa = 1$, BFGP would neglect the other objective.

As we will show, by controlling the way how p_j is calculated, we can achieve the objectives of *energy saving* and *load balance* with an algorithm originally designed for the common building block problem of minimizing the weighted sum of deployment cost for the given components (i.e., BFGP).

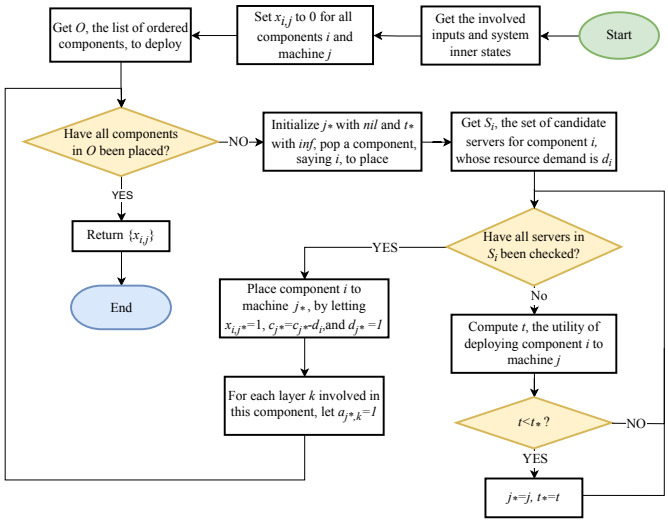


Fig. 2: The block diagram of BFGP specified by Algorithm 1.

B. Algorithm Details

Algorithm 1 along with the block diagram of Figure 2 sketches the workflow of the proposed BFGP (Best Fit Greedy Placement) algorithm, which is able to achieve optimized deployment cost for the common building block problem of models (6) and (8).

Basically, BFGP places function components one-by-one (Line 5); and for the placement of each component, BFGP greedily embeds it at the server that introduces the minimum additional deployment cost (Lines 8-12). Here, the deployment cost is evaluated by the tuple of $\langle \max(0, d_i - c_j), \sum_{k \in L_i} w_{j,k}(1 - a_{j,k}) \rangle$ rather than $\sum_{k \in L_i} w_{j,k}(1 - a_{j,k})$. This is to handle the case when there does not exist a deployment that satisfies all their resource demands without overloading servers. With such a design, BFGP would let the system admit all deployment requests and balance their loads among all available servers in such situations. Given that it would be much easier to find a server with sufficient capacity to host a component with small resource demands, BFGP processes components in non-increasing order of their demands (Line 4). Moreover, as §III-B has discussed, by configuring $w_{i,j}$ values appropriately, BFGP supports hardware heterogeneity.

As Algorithm 2 shows, the implementation of $getS()$ is the key to achieving the objectives of *energy saving* and *load balance* with BFGP. Here, $H()$ is the *Heaviside* function as (9) defines. When energy saving is desired, Algorithm 2 prefers to select the top-k heaviest-loaded servers with enough remaining resources to host the component (i.e, BFGP-ES); while when *load balance* is pursued, the top-k lightest-loaded servers are preferred (i.e., BFGP-LB). Besides, we also consider the design that $getS(d_i, S_i)$ directly returns S_i, purely pursuing the goal of deployment cost minimization without concerning

Algorithm 1: Best Fit Greedy Placement (BFGP)

Input: $\{d_i\}, \{S_i\}, \{L_i\}$;
Data: $\{a_{j,k}\}, \{b_j\}, \{c_j\}, \{w_{j,k}\}$;
Output: $\{x_{i,j}\}$;

foreach $i \leftarrow 1, 2, \cdots, n$ **do**
  foreach $j \leftarrow 1, 2, \cdots, m$ **do**
    $x_{i,j} \leftarrow 0$;
$O \leftarrow getOrderedComponentsToDeploy()$;
/* in non-increasing order of their d_is */
foreach $i \in O$ **do**
  $j_* \leftarrow nil$; $t_* \leftarrow +\infty$;
  $\hat{S}_i \leftarrow getS(d_i, S_i)$; /* get candidate servers */
  foreach $j \in \hat{S}_i$ **do**
    $t \leftarrow \langle \max(0, d_i - c_j), \sum_{k \in L_i} w_{j,k}(1 - a_{j,k}) \rangle$;
    if $t < t_*$ **then**
      $j_* \leftarrow j$; $t_* \leftarrow t$;
  $x_{i,j_*} \leftarrow 1$; $c_{j_*} \leftarrow c_{j_*} - d_i$; $b_{j_*} \leftarrow 1$;
  foreach $k \in L_i$ **do**
    $a_{j_*,k} \leftarrow 1$;
return $\{x_{i,j}\}$;

Algorithm 2: The Implementation of $getS()$

Input: d_i, S_i;
Data: $\{b_j\}, \{g_j\}, \kappa, policy, \{c_j\}$;
Output: $\hat{S}_i$;
/* $H()$ is the *Heaviside* function of (9) */
if *policy is Pure_Cost_Minimimization* **then**
  /* BFGP-Full */
  return S_i;
else if *policy is Energy_Saving* **then**
  /* BFGP-ES */
  foreach $j \in S_i$ **do**
    $p_j \leftarrow \langle H(c_j - d_i), b_j, \frac{c_j}{g_j} \rangle$;
else if *policy is Load_Balance* **then**
  /* BFGP-LB */
  foreach $j \in S_i$ **do**
    $p_j \leftarrow \left\langle H(c_j - d_i), \frac{c_j}{\bar{c}_j}, b_j \right\rangle$;
$k \leftarrow \lfloor \kappa |S_i| \rfloor$; /* $0 < \kappa \leq 1$ */
$\hat{S}_i \leftarrow$ select the top-k servers in S_i respecting their p_js;
return $\hat{S}_i$;

operational goals (i.e., BFGP-Full).

$$H(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}. \tag{9}$$

C. Time Complexity

Obviously, given m servers, it is easy for Algorithm 2 to obtain the top-(κm) candidate servers within $O(m^2)$ times, where $0 < \kappa \leq 1$. Based on this, the worst-case time complexity of BFGP is about $O(n \log n + n(m^2 + lm + l))$. According to the property of O-notation [29], we would have $O(n \log n + n(m^2 + lm + l)) = O(n \log n + n(m^2 + ml)) = O(n(\log n + m(m + l)))$. Since $m(m + l)$ is generally larger than $\log n$, we would further obtain $O(n(\log n + m(m + l)) = O(nm(m + l))$ for the proposed BFGP algorithm.

V. Performance Evaluation

To study the performance of BFGP, in this section, we use it to deploy synthesized containerized components and analyze the deployment cost, computation time, proportions of used/active servers, and the maximum server load in detail. Extensive results indicate that the proposed BFGP is very flexible and efficient: it is able to achieve near-optimal placement of containerized components in terms of the deployment cost very efficiently (within $O(nm(m + l))$ time) with the variant of BFGP-Full, and is expressive to achieve high-level policies like *energy saving* and *load balance* with the variants of BFGP-ES and BFGP-LB, respectively.

A. Methodology

Workloads. According to the design, our proposed algorithms can work properly for the case of deploying any number (i.e., n) of components to any number (i.e., m) of servers. We find that despite the various absolute result values obtained, consistent conclusions are observed when the relationship between m and n changes. Thus, without loss of generality and motivated by related works [16], we consider the case of deploying n containerized function components into a cluster with n servers (i.e., $m = n$). For each component, both the number of its involved container image layers and the size of each layer are randomly generated following the distributions obtained from Docker Hub, a popular public image registry [31]. In practice, different components might share layers. We suppose that all these n components involve L_t layers in total, which are made up of L_d diverse layers. Here, both L_t and L_d are configurable parameters used in tests. We define the value of $1 - \frac{L_d}{L_t}$ as sr, i.e., the *sharing ratio* of layers between components. In tests, the total number of distinct layers in the cluster (denoted by L_d) is generated by $L_t * (1 - sr)$. For each server, we further assume that it already hosts $\lfloor sr * L_d \rfloor$ randomly selected image layers. As for the cluster loads, we assume that each server is with the capacity of $\bar{c}$ and the demands of components follow the truncated uniform distribution of $\bar{c} \min(1, \lambda U[0.9, 1.1])$, where λ is a controllable parameter and $\lambda \bar{c}$ denotes the average resource demand of a component and it can be placed on any server. Regarding the current server states, we consider that $b^{\#}$ of them are active with an average load of b^* before the deployment. For the j-th server, its activation energy cost is g_i. By default, the values of n, sr, λ, $\bar{c}$, κ, $b^{\#}$, b^*, and g_j are set to 100, 0.2, 0.2, 100, 0.3, 0, 0, and 1, respectively. In tests, we vary the values of these parameters to study their impacts on the deployment cost, computation time, proportions of used/active servers, and maximum server load. For each parameter setting, we run 20 trials and calculate both their average, minimum, and maximum values to report.

Metrics. We mainly use the deployment cost calculated by Equation (10), i.e., the weighted sum of the total container image volumes to transfer. Here, B is a large penalty factor with a default value of 10^{10}. As the level of network interference can be captured by the amount of triggered traffic, thus, we can use the defined "cost value", computed by E.q. (10), as the metric of network interference. In some cases, we also record the running time for the study of algorithm efficiency, the proportion of used servers for the evaluation of *energy saving*, and the maximum server load for the evaluation of *load balance*. Our tests are conducted on a 64-bit Ubuntu 20.04 server equipped with 32GB RAM and one Intel(R) Core(TM) i7-8700 (3.20GHz) CPU.

$$cost(\{x_{i,j}\},\{y_{j,k}\}) = \sum_{j=1}^{m}\sum_{k=1}^{l} w_{j,k}y_{j,k} + B\max(0, \max_{j}\frac{\sum_{i=1}^{n} d_i x_{i,j} - c_j}{\bar{c}_j}) \tag{10}$$

Baselines. As we have analyzed in Section II-C, existing schemes either have not made usage of the partial reusability of layers among different containerized components [10, 19], or do not support the changeable operational policies of *energy saving* and *load balance* [16, 17, 21–23]. Thus, in tests, we mainly consider the following schemes for performance study:

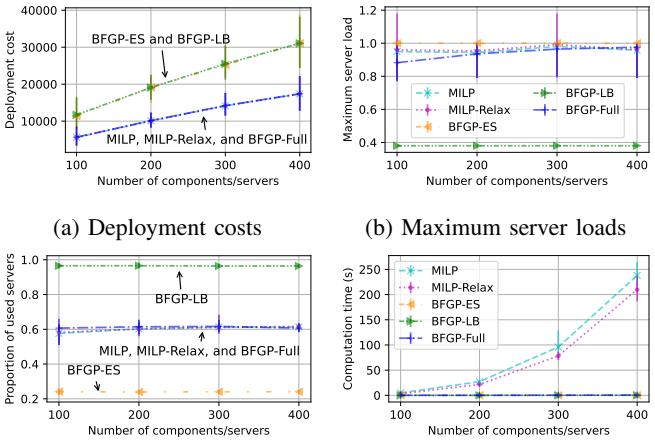
- **MILP**: components are deployed according to the optimal results of the augmented model of (11), which is a Mixed-Integer Linear Programming (MILP);
- **MILP-Relax**: components are deployed following the rounded results of the relaxation of model (11);
- **BFGP-Full**: variant of BFGP, in which $getS(d_i, S_i)$ returns S_i, purely pursuing the goal of cost minimization;
- **BFGP-ES**: variant of BFGP, respecting the operational goal of *energy saving*;
- **BFGP-LB**: variant of BFGP, respecting the operational goal of *load balance*.

$$\text{Minimize } cost(\{x_{j,k}\},\{y_{j,k}\}) \quad s.t.\ (1),(4),(5),(10). \tag{11}$$

All the above schemes are implemented in Python 3 and employ single-threaded designs. For both MILP and MILP-Relax, the involved models are solved with the commercial Mosek solver with default settings [32].

B. Results

BFGP is expressive. Figure 3 shows the performance of deploying n components to n servers, under the control of different schemes, when n increases from 100 to 400. Results in Figure 3a indicate that when the operational goal of either energy saving or load balance is not considered, compared with the optimal deployments conducted by MILP, both BFGP-Full and MILP-Relax achieve near-optimal deployment costs. Even though pursuing totally different operational goals, BFGP-ES and BFGP-LB obtain almost the same deployment costs ($gap \leq 3\%$), which are about $1.8 - 2.1\times$ of those of MILP. Regarding the proportion of used servers and their maximum load shown in Figures 3c and 3b, as expected, BFGP-ES and BFGP-LB are the best solutions, respectively.



(a) Deployment costs (b) Maximum server loads

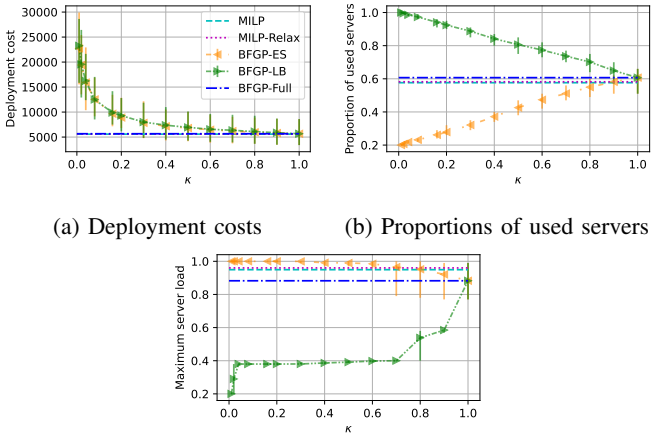
(c) Proportions of used servers (d) Computation time

Fig. 3: **[Effectiveness and Expressiveness]** Compared with MILP, when the operational goals like *energy saving* or *load balance* are not considered, both BFGP-Full and MILP-Relax achieve near-optimal deployment costs. Moreover, BFGP-ES and BFGP-LB obtain the best performance respecting their design goals, respectively, showing that our BFGP algorithm is expressive and flexible. Also, as expected, BFGP is very efficient as all its variants are able to obtain results for large-scale deployment tasks within tens or hundreds of milliseconds.

For example, to host 200 components, BFGP-ES employs about 24% of the servers with a maximum load of 100%, while BFGP-LB uses almost all servers with a balanced load smaller than 40%. As a comparison, due to the unawareness of the operational goal, MILP, MILP-Relax, and BFGP-Full occupy about 60% of the servers with highly-skewed loads, as their maximum server load approximates 100%.

BFGP is efficient. Figure d shows the computation times these algorithms take. Roughly, with the increase of problem scale, the time costs of MILP grow exponentially, up to about 240s per instance when m reaches 400. While BFGP-ES, BFGP-LB, and BFGP-Full are very fast, taking less than about 0.08s, 0.09s, and 0.55s, respectively. Theoretically, as it is built upon linear programming, MILP-Relax is able to obtain its solutions within polynomial time by using off-the-shelf high-performance solvers. However, we observe that its LP model generally involves a large number of variables and constraints, growing super-linear with the number of components and servers. For instance, when deploying 200 components, the model involves about hundreds of thousands of variables and constraints, which increases to millions when the problem scale grows up to 400. We also notice that, in some cases, due to the relax-then-round designs, some servers' loads under the control of MILP-Relax are likely to exceed 100% even though there are still servers with enough capacities. Accordingly, those heuristic designs built upon LP relaxation do not suit our problems. All the above results imply that the proposed BFGP is not only very effective, but also expressive to support the operational goals of energy saving and load balance.

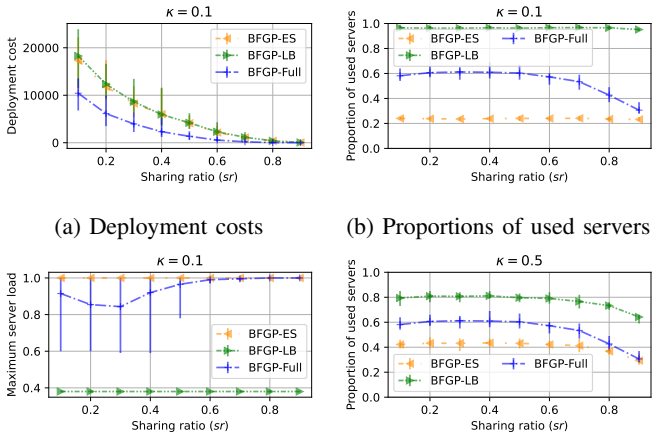
Impacts of κ. To study the expressiveness of BFGP, we



(a) Deployment costs (b) Proportions of used servers

(c) Maximum server loads

Fig. 4: **[Impacts of κ]** There are trade-offs among the deployment cost, the proportion of used servers, and maximum server load. With the increase of κ, both BFGP-ES and BFGP-LB would degenerate to BFGP-Full. By tuning κ, both BFGP-ES and BFGP-LB would achieve the desired high-level deployment policies of *energy saving* and *load balance* according to their designs, with the cost of higher yet controllable deployment costs.



(a) Deployment costs (b) Proportions of used servers

(c) Maximum server loads (d) Proportions of used servers

Fig. 5: **[Impacts of sr]** A higher sharing ratio results in less deployment costs for BFGP-Full, BFGP-ES, and BFGP-LB, with the price of slightly increased proportions of used servers.

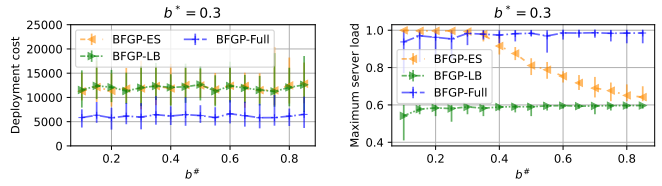
vary κ, the ratio of servers selected from the candidates for the deployment of BFGP-ES and BFGP-LB, then record its impacts on the deployment cost, maximum server load, and proportion of used servers. Predictably, as Figures 4b and 4c show, when $\kappa|S_i| = 1$, BFGP-ES and BFGP-LB achieve their best performances, in terms of the proportion of used servers, about 20% for BFGP-ES, and maximum server load, about 20% for BFGP-LB, respectively. In such settings, they ignore the goal of minimizing the deployment cost, leading to very high deployment costs. As Figure 4a shows, with the increase of κ, their deployment costs decrease significantly and finally approximate those of MILP, MILP-Relax, and BFGP-Full when $\kappa = 1$. However, results in Figures 4b and 4c also imply that the improvement in the deployment cost is with the price of more used servers for BFGP-ES, and higher maximum server load for BFGP-LB. Interestingly, for BFGP-ES, the growth of the proportion of used servers is likely to be linear, while for BFGP-LB, the pattern of the increase of the maximum server load can be split into three stages: it 1) firstly increases from about 0.2 to 0.38 almost exponentially before κ reaches 0.04, then 2) grows very slowly and linearly to 0.4 until $\kappa = 0.7$, and 3) finally rises to meet with the result of BFGP-Full at $\kappa = 1$.

We also observe that the proportion of used servers for BFGP-LB and the maximum server load for BFGP-ES also drop with the increase of κ and finally meet with the results of BFGP-Full. The above results show that there are trade-offs among the deployment cost, proportion of used servers, and maximum server load, among which κ is the key to control. By tuning the value of κ, both BFGP-ES and BFGP-LB can make a balance between their two optimization goals detailed in (6) and (8) with respect to the use instances. The law of κ's impact depends on the cluster configurations and workload patterns and in the cases studied here, just letting $\kappa = 0.1$ yields good balances between the introduced benefits and penalties of deployment cost for both BFGP-ES and BFGP-LB.

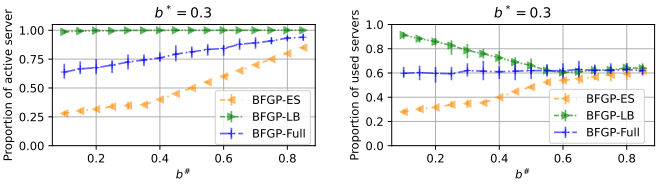
In all these tests, the performances of BFGP-Full are very close to those of MILP and MILP-Relax. As the solving of both MILP and MILP-Relax is time-costly, in the following study, we mainly use BFGP-Full as the baseline.

Impacts of sr. By default, the value of the sharing ratio parameter is set to 0.2. To investigate its impact, we vary the sr value from 0.1 to 0.9. As Figure 5 shows and as expected, a larger sharing ratio would result in less deployment cost for all the test schemes; yet, for both BFGP-ES and BFGP-LB, the level of sharing has few impacts on their proportions of used servers and maximum server loads. In consideration of the fact that more and more advanced layer optimization techniques are on their way [18], we argue that the proposed BFGP algorithm(s) would bring increased benefits to future applications. Also, it shows that, for BFGP-Full, the growth of sr leads to a slight decrease in their proportion of used servers and increases in the maximum load. This is reasonable. Recall that BFGP-Full repeatedly places each component to the server introducing the minimum deployment cost; when the sharing ratio is high, minimizing the deployment cost leads to the goal of content-aware component consolidation, yielding reduced used servers and increased maximum server loads. As shown in Figure 4, with the increase of κ, both BFGP-ES and BFGP-LB will degenerate to BFGP-Full. Thus, we further test the behaviors of BFGP-ES and BFGP-LB by increasing κ to 0.5; the results in Figure 5d confirm our analysis.

Impacts of $b^{\#}$. Next, we look into the case in which $b^{\#}m$ randomly selected servers are already active and are hosting other applications with an average load of b^*. In tests, the loads of pre-active servers are randomly generated following the uniform distribution of $[0, 0.6]$ (i.e., $b^* = 0.3$), and we vary the value of $b^{\#}$ from 0.1 to 0.8. Figure 6 shows the impact

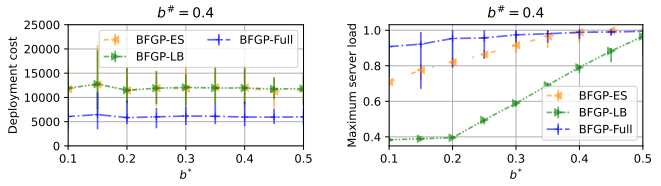


(a) Deployment costs (b) Maximum server loads

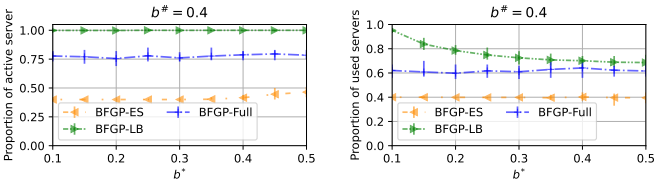


(c) Proportions of active servers (d) Proportions of used servers

Fig. 6: **[Impacts of $b^\#$]** The ratio of pre-active servers has few impacts on the deployment cost; while there is still room for the optimization of the proportion of used servers for BFGP-LB under the operational goal of load balance.



(a) Deployment costs (b) Maximum server loads



(c) Proportions of active servers (d) Proportions of used servers

Fig. 7: **[Impacts of b^*]** The loads of pre-active servers have few impacts on the deployment cost; while there is still room for optimizing maximum server load for BFGP-LB under the operational goal of energy saving.

of $b^\#$. As Figure 6a implies, for all schemes, the deployment costs remain stable with the change of $b^\#$, i.e., about 6000 for BFGP-Full, and about 11900 for both BFGP-ES and BFGP-LB, respectively. Regarding the maximum server load, as shown in Figure 6b, the results of BFGP-Full and BFPG-LB keep consistent values of almost 1, and 0.6, respectively. However, for BFGP-ES, more pre-active servers would lead to a smaller maximum server load—with the value of $b^\#$ increasing from 0.1 to 0.9, the maximum server load under the schedule of BFGP-ES decrease from the high value of about 1 to approximate the results of BFGP-LB, i.e., 0.6. This is consistent with the design of BFGP-ES: it would prefer to select servers that are already active or used as candidates in Algorithm 2; accordingly, with the growth of $b^\#$, BFGP-ES would evolve into a design close to that of BFGP-LB.

As expected, the results in Figure 6c also show that, for both BFGP-ES and BFGP-Full, the proportion of active servers after the deployment is increasing with $b^\#$, starting from about 0.25 and 0.63 to about 0.83 and 0.95, respectively. However, as shown in Figure 6d, for BFGP-LB, the proportion of servers that are actually used to host these n components decreases, from about 0.9 to about 0.6, and finally approximates the results achieved by BFGP-Full. Such results imply that, for the operational goal of load balance, there is still room to reduce the number of used servers without enlarging the maximum server load. We leave this as future work. Regarding BFGP-ES, the proportion of its actually used servers also increases slowly from about 0.3 to about 0.6, since these pre-active servers already host applications with an average load of b^*, thus the usage of more pre-active servers leads to a larger number of used servers. As for BFGP-Full, the proportion of actually used servers remains stable since these pre-active servers are not heavy-loaded.

Impacts of b^*. Last but not least, we also change b^*, the average load of these pre-active servers, to study its impact on the deployment performances. To highlight the comparison, in tests, we fix the value of $b^\#$ to 0.4 and increase b^* from 0.1 to 0.5. As Figure 7a shows, like the increase of $b^\#$, the deployment costs achieved by all schemes remain stable. And in this case, the maximum server load would increase, even for BFGP-LB. Indeed, confirmed by Figure 7b, the growth of maximum server load is mainly caused by our workload settings: for each pre-active serve, its load is randomly synthesized following the uniform distribution of $U[0, 2b^*]$, which will dominate the maximum server load of the entire cluster in the tests once $b^* \geq 0.2$. In addition, according to the current design of Algorithm 2, BFGP-LB will prefer to select pre-active servers as candidates, this results in a reduced proportion of used servers with the growth of b^* as Figure 7d shows. We also find that, as only 40% of the servers are configured to be active, the proportions of active servers remain stable for BFGP-Full and BFGP-LB. Finally, like the case of BFGP-LB shown in Figure 6, results show that, for BFGP-ES, with the increase of b^*, even though the maximum server load increases (Figure 7b), the proportion of actually used servers remains almost the same (Figure 7d), suggesting that there is still room for optimizing their maximum server load under the operational goal of energy saving. We leave this as future work.

VI. Related Work

Our work is to optimize the deployment of serverless applications with content-aware component placement. To put it in context, we start with describing related research on *registry enhancement*, *layer pre-distribution*, *p2p layer delivery*, and *on-demand layer transmission*, respectively.

Registry enhancement. In production, the centralized container image registry is likely to be the bottleneck of the entire system. To scale the registry out, Anwar *et al* optimize registry clusters with techniques like two-level cache and

layer prefetching, based on the skewed nature of the pull and push workloads observed in IBM's clouds [33]. Likewise, Zhao *et al* analyzed the structures of the container layers and images hosted on the public registry of Docker hub [31], and designed a flexible performance-optimized deduplication Docker registry architecture by exploiting the observed high file redundancy among public container layers [34]. Different from them, CoMICon[35] develops a cooperative registry system for Docker, enabling nodes to conduct distribution pulls of the same image, thus reducing the service start times.

Layer pre-distribution. To migrate the bottleneck introduced by the centralized registry, researchers have explored the idea of pre-distributing container layers to selected servers to enable parallel pull. For instance, Smet *et al* looked into the optimization of placing container layers at strategic storage nodes, such that more service deploy requests could get served within the desired response time in edge computing [36]. Likewise, Darrous *et al* studied the problem of pre-dispersing layers and images across selected servers to reduce the maximum retrieval time of an image to any other edge-server [37]. Although pre-dispersing container layers among the network would help, the designs involved in [36] and [37] rely on an unrealistic implicit assumption that the details of future service deploy requests are known in advance. Moreover, to accelerate the retrieval upon pre-dispersing, they both suggest the optimization of downloading different layers of a container from various strategic nodes in parallel [36, 37], a non-trivial feature that has not been supported by off-the-shelf container runtime systems like Docker and LXC.

P2P layer delivery. Another widely considered optimization for the deployment of containerized service is to accelerate the delivery of missed container layers. Motivated by the design and success of peer-to-peer (P2P) file-sharing systems, many enterprises and cloud providers have employed P2P techniques like BitTorrent and its varieties to achieve fast dissemination of container images in large-scale data centers. For example, Uber and Alibaba have designed and opened source their production solutions named Kraken [13] and Dragonfly [14], respectively. Different from the problem of content-aware service placement, these P2P solutions mainly focus on how to accelerate the process of a given pull efficiently.

On-demand layer transmission. Based on the observations that the start of a container usually only depends on a small part of the entire container images, techniques like remote image along with on-demand layer transmission have been proposed. For example, Slacker employs centralized shared storage to store layers for all images. With customized storage drivers, works in Slacker can quickly provision container layers to minimize the startup delay by lazily fetching [7]. To further develop this idea, Dadi even redesigns the structure of container layers and builds a block-level image service [15].

Orthogonal with the above studies, the idea we explore in this paper is to reduce the amount that it takes for the pull of docker images with content-aware service placement. Indeed, all the above techniques can be integrated with our proposed content-aware deployment. Recently, there are two very related works, [16] and [17], that also explicitly exploit the layered and shareable structures of docker images in service deployment to speed up the service provisioning and reduce storage consumption. However, their proposed algorithms are less expressive since they do not support high-level operational policies like energy saving and load balance.

VII. Conclusion and Future Work

Nowadays, container-based serverless computing has emerged as the new paradigm of cloud computing. This paper studies the fundamental problem of how to place containerized function components involved in serverless computing is the best, in terms of both the deployment cost (e.g., startup delay and the inter-service/application network interference) and operational policies like energy saving and load balance. We prove that the problem is NP-hard and design an efficient yet flexible heuristic solution named BFGP to solve it. Extensive simulation results show that the proposed BFGP not only achieves near-optimal placements but also supports policies like energy saving and load balance.

For future work, we plan to make a co-design of the placement of containerized components and the delivery of missing layers, such that the delay of service deployment could be further reduced. Particularly, in some cases, a layer absent at a server might be held by multiple other servers, and on the contrary, different servers also might need to fetch the same missing layer; accordingly, it is promising to employ advanced transport protocols/solutions to achieve efficient transmissions of layers intelligently.

References

[1] P. Castro, V. Ishakian *et al.*, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019.

[2] S. Kounev, N. Herbst *et al.*, "Serverless computing: What it is, and what it is not?" *Communications of the ACM*, vol. 66, no. 9, p. 80–92, aug 2023.

[3] N. Gottlieb, "State of the serverless community survey results," 2020, https://www.serverless.com/blog/state-of-serverless-community/ [Online; accessed 18-Dec-2020].

[4] J. Jiang, S. Gan *et al.*, "Towards demystifying serverless machine learning training," in *Proceedings of the ACM SIGMOD*, June 2021, pp. 857–871.

[5] E. Oakes, L. Yang *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proceedings of the USENIX ATC*, Boston, MA, Jul. 2018, pp. 57–70.

[6] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, nov 2022.

[7] T. Harter, B. Salmon *et al.*, "Slacker: Fast distribution with lazy docker containers," in *Proceedings of the USENIX FAST*, Santa Clara, CA, Feb. 2016, pp. 181–195.

[8] I. E. Akkus, R. Chen *et al.*, "SAND: Towards high-performance serverless computing," in *Proceedings of the USENIX ATC*, Boston, MA, Jul. 2018, pp. 923–935.

[9] A. Klimovic, Y. Wang *et al.*, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proceedings of the USENIX OSDI*, Carlsbad, CA, Oct. 2018, pp. 427–444.
[10] M. Shahrad, R. Fonseca *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the USENIX ATC*, Jul. 2020, pp. 205–218.
[11] S. Luo, T. Ma *et al.*, "Efficient multisource data delivery in edge cloud with rateless parallel push," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10 495–10 510, 2020.
[12] S. Luo, H. Yu *et al.*, "Efficient file dissemination in data center networks with priority-based adaptive multicast," vol. 38, no. 6, pp. 1161–1175, 2020.
[13] "Kraken," 2020, https://github.com/uber/kraken [Online; accessed 06-Dec-2020].
[14] "Dragonfly," 2020, https://github.com/dragonflyoss/Dragonfly [Online; accessed 06-Dec-2020].
[15] H. Li, Y. Yuan *et al.*, "DADI: Block-level image service for agile and elastic application deployment," in *Proceedings of the USENIX ATC*, Jul. 2020, pp. 727–740.
[16] L. Gu, D. Zeng *et al.*, "Exploring layered container structure for cost efficient microservice deployment," in *Proceedings of the IEEE INFOCOM*, 2021, pp. 1–9.
[17] L. Gu, D. Zeng *et al.*, "Layer aware microservice placement and request scheduling at the edge," in *Proceedings of the IEEE INFOCOM*, 2021, pp. 1–9.
[18] D. Skourtis, L. Rupprecht *et al.*, "Carving perfect layers out of docker images," in *Proceedings of the 11th HotCloud*, USA, 2019, p. 17.
[19] H. Yu, A. A. Irissappane *et al.*, "Faasrank: Learning to schedule functions in serverless platforms," in *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2021, pp. 31–40.
[20] Z. Li, L. Guo *et al.*, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys*, vol. 54, no. 10s, sep 2022.
[21] L. Gu, Z. Chen *et al.*, "Layer-aware collaborative microservice deployment toward maximal edge throughput," in *Proceedings of the IEEE INFOCOM*, 2022, pp. 71–79.
[22] D. Zeng, H. Geng *et al.*, "Layered structure aware dependent microservice placement toward cost efficient edge clouds," in *Proceedings of the IEEE INFOCOM*, 2023, pp. 1–9.
[23] J. Lou, H. Luo *et al.*, "Efficient container assignment and layer sequencing in edge computing," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1118–1131, 2023.
[24] C. Guo, H. Wu *et al.*, "Rdma over commodity ethernet at scale," in *Proceedings of the ACM SIGCOMM Conference*, New York, NY, USA, 2016, p. 202–215.
[25] T. Mastelic, A. Oleksiak *et al.*, "Cloud computing: Survey on energy efficiency," *ACM Computing Surveys*, vol. 47, no. 2, dec 2014.
[26] H. Shen, H. Wang *et al.*, "An instability-resilient renewable energy allocation system for a cloud datacenter," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 1020–1034, 2023.
[27] P. Sharma, "Challenges and opportunities in sustainable serverless computing," *SIGENERGY Energy Inform. Rev.*, vol. 3, no. 3, p. 53–58, oct 2023.
[28] B. Acun, B. Lee *et al.*, "Carbon explorer: A holistic framework for designing carbon aware datacenters," in *Proceedings of the 28th ACM ASPLOS*, New York, NY, USA, 2023, p. 118–132.
[29] T. H. Cormen, C. E. Leiserson *et al.*, *Introduction to algorithms*. MIT press, 2009.
[30] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, dec 2022.
[31] N. Zhao, V. Tarasov *et al.*, "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, 2021.
[32] M. ApS, *MOSEK Optimizer API for Python 9.2.49*, 2021.
[33] A. Anwar, M. Mohamed *et al.*, "Improving docker registry design based on production workload analysis," in *Proceedings of the USENIX FAST*, Oakland, CA, Feb. 2018, pp. 265–278.
[34] N. Zhao, H. Albahar *et al.*, "Duphunter: Flexible high-performance deduplication for docker registries," in *Proceedings of the USENIX ATC*, Jul. 2020, pp. 769–783.
[35] S. Nathan, R. Ghosh *et al.*, "Comicon: A co-operative management system for docker container images," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2017, pp. 116–126.
[36] P. Smet, B. Dhoedt, and P. Simoens, "Docker layer placement for on-demand provisioning of services on edge clouds," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 1161–1174, 2018.
[37] J. Darrous, T. Lambert, and S. Ibrahim, "On the importance of container image placement for service provisioning in the edge," in *Proceedings of the 28th International Conference on Computer Communication and Networks (ICCCN)*, 2019, pp. 1–9.