






# SLO-Aware Function Placement for Serverless Workflows With Layer-Wise Memory Sharing

Dazhao Cheng , Senior Member, IEEE, Kai Yan , Xinquan Cai , Yili Gong , and Chuang Hu , Member, IEEE

**Abstract**—Function-as-a-Service (FaaS) is a promising cloud computing model known for its scalability and elasticity. In various application domains, FaaS workflows have been widely adopted to manage user requests and complete computational tasks efficiently. Motivated by the fact that function containers collaboratively use the image layer’s memory, co-placing functions would leverage memory sharing to reduce cluster memory footprint, this article studies layer-wise memory sharing for serverless functions. We find that overwhelming memory sharing by placing containers in the same cluster machine may lead to performance deterioration and Service Level Objective (SLO) violations due to the increased CPU pressure. We investigate how to maximally reduce cluster memory footprint via layer-wise memory sharing for serverless workflows while guaranteeing their SLO. First, we study the container memory sharing problem under serverless workflows with a static Directed Acyclic Graph (DAG) structure. We prove it is NP-Hard and propose a 2-approximation algorithm, namely MDP. Then we consider workflows with dynamic DAG structure scenarios, where the memory sharing problem is also NP-Hard. We design a Greedy-based algorithm called GSP to address this issue. We implement a carefully designed prototype on the OpenWhisk platform, and our evaluation results demonstrate that both MDP and GSP achieve a balanced and satisfying state, effectively reducing up to 63% of cache memory usage while guaranteeing serverless workflow SLO.

**Index Terms**—Serverless computing, serverless cluster, container placement, scheduling.

## I. INTRODUCTION

FUNCTION as a Service (FaaS) stands at the forefront of cloud computing models, providing developers with a revolutionary approach to code deployment. This innovative paradigm allows developers to focus exclusively on crafting and deploying individual code functions, liberating them from the complexities of managing underlying infrastructure [1]. Commonly referred to as serverless computing, FaaS simplifies infrastructure management and facilitates code execution within a serverless environment. Its allure lies in on-demand scalability and pay-per-use pricing, making it particularly appealing for applications with unpredictable or varying workloads. Platforms

such as AWS Lambda [2] and Azure Functions [3] exemplify FaaS, offering developers the capacity to leverage multiple CPU cores and virtually unlimited memory for efficient execution of large-scale functions.

Despite its myriad advantages, FaaS presents challenges, notably in addressing concerns like eliminating cold starts, function execution scheduling, and resource provisioning [4], [5]. Efficient resource management stands out as a critical concern, enhancing function performance while minimizing resource wastage for cloud vendors [6].

In this landscape, memory emerges as a precious resource for serverless platforms. Excessive memory contention gives rise to issues like cold starts and execution delays. Notably, despite previous research delving into computational resource provisioning for serverless functions and workflows, the potential of memory sharing between function execution sandboxes, such as memory sharing between Docker [7] containers, has been largely overlooked. Additionally, there is a notable absence of function placement algorithms considering memory sharing while ensuring the Service Level Objective (SLO) of workflows.

Functions are executed in *containers*, which are built from *images* consisting of *layers*. Functions sharing the memory of layer file caches could optimize memory resource utilization significantly. Docker, utilizing OverlayFS [8] as the underlying file system for storing image files, provides an opportunity to share function container memory by layer. This paper explores the *layer-wise memory sharing for serverless functions*.

The focus is on FaaS workflows, where functions exhibit strong layer relations and are organized in specific patterns [9], [10], [11], [12]. Through experiments conducted on OpenWhisk [13], our paper demonstrates that actively leveraging layer-based function memory sharing can yield substantial benefits for FaaS workflows. Placing related function containers on the same work-node (machine) reduces the overall memory footprint for the serverless cluster due to the inherent memory sharing feature of containers. However, it is crucial to consider CPU pressure to avoid workflow execution time degradation. Excessive pressure on the work-node may lead to increased function execution time, potentially violating workflow SLOs. Hence, effectively managing the trade-off between memory sharing and SLO guaranteeing emerges as a vital aspect to reduce the resource footprint while ensuring optimal performance.

In this paper, we investigate how to reduce memory consumption by maximizing memory sharing for serverless workflows while guaranteeing their SLOs. We investigate the bottom features of docker’s ability for cache memory sharing and discussed

Manuscript received 6 December 2023; revised 8 March 2024; accepted 17 April 2024. Date of publication 22 April 2024; date of current version 3 May 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFE0205700, in part by the National Natural Science Foundation of China under Grant 62341410 and under Grant 62302348, and in part by the General Program of Hubei Provincial Natural Science Foundation of China under Grant 2023AFB831. Recommended for acceptance by D. Tiwari. (Corresponding authors: Chuang Hu; Yili Gong.)

The authors are with the School of Computer Science, Wuhan University, Hubei 430072, China (e-mail: dcheng@whu.edu.cn; yankai@whu.edu.cn; xinquancai@whu.edu.cn; yiligong@whu.edu.cn; handc@whu.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3391858

the concern of safety. We then propose a novel mathematical model that takes into account the structure of the workflow's Directed Acyclic Graph (DAG), the workload on work-nodes, and function characteristics. Based on this model, we formulate an optimization problem that is proven to be NP-Hard, akin to the multiple knapsack problem [14]. Drawing on heuristic insights, we introduce two function placement algorithms, namely MDP (Merged DAG Partition) and GSP (Greedy Step Placement), to tackle function placement within serverless clusters. They both leverage memory sharing while ensuring the workflow's SLO to address the container placement problem for both static and dynamic DAGs. MDP is a 2-approximation algorithm for static DAGs. It leverages the sharing potential of function image layers to merge nodes of a DAG into different sub-graphs. Containers placement decision within the cluster is made based on those sub-graphs. Then we consider workflows with dynamic DAG structure scenarios and design a Greedy-based algorithm called GSP. GSP leverages the commonality and parallelism among functions of the same step/stage [6] to handle the dynamic scenarios of workflows. A well-designed system implementing algorithms above is built upon OpenWhisk [13]. Finally we conduct extensive experiments on the OpenWhisk platform to evaluate the performance of our algorithms. The experimental results demonstrate that our algorithms achieve an effective balance between memory sharing and SLO guaranteeing, and importantly, the placement decisions are made with minimal overhead.

This paper makes the following key contributions:

- Illustrating the potential and benefits of memory sharing among serverless function containers, specifically Docker containers, while explicitly considering the trade-off between sharing and SLO-guaranteeing.
- Formulating the function placement problem as NP-Hard, considering the metrics of DAGs, functions, and work-nodes. The paper proposes two heuristic solutions tailored for static and dynamic DAGs.
- Implementing several carefully designed modules on the OpenWhisk platform to put the algorithms into service.
- Conducting extensive experiments on OpenWhisk to validate the effectiveness of the proposed algorithms and showcase their significant impact on FaaS workflows concerning memory sharing and SLO guaranteeing.

## II. BACKGROUND AND MOTIVATION

### A. OverlayFS and Container Memory Sharing

In FaaS platforms, functions are usually executed in sandboxes [2], [3], [13], [15], which are isolated and controlled environments that ensure security, reliability, and scalability. Docker [7] containers are often used as the underlying technology for creating and managing these sandboxes (e.g., OpenWhisk [13]). Containers are running instances of *images*. An *image* serves as a self-contained and executable software package formed by *image layers* [16].

The latest versions of Docker employ OverlayFS [8] as the underlying file system for storing image layers. It has come to our attention that *OverlayFS supports page cache sharing* [16],

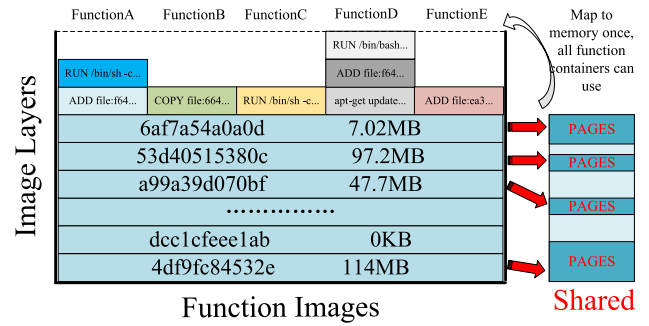


Fig. 1. Decomposition of a workflow's images, different functions share plenty of layers. Their memory page caches can be shared among functions.

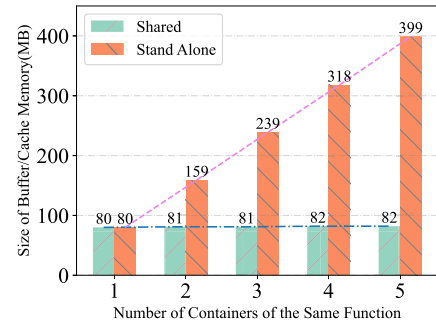


Fig. 2. Buffer/Cache memory usage after spawning containers.

meaning that an image layer file page only needs to be mapped to memory once, and all containers using it can access the same page (Fig. 1). Out of curiosity, we wanted to investigate how OverlayFS page cache sharing works in FaaS platforms. Therefore, we conducted a simple experiment on OpenWhisk by spawning five containers of the same function on the same machine or on different machines. The results, as shown in Fig. 2, reveal that the Buffer/Cache memory size only significantly increases once when five containers are placed together (Shared). However, if the five containers are spawned on different machines, they consume multiple portions of Buffer/Cache memory (Stand Alone). This indicates that OverlayFS does reduce serverless cluster's overall memory footprint for containers, as page caches of image layers are shared if containers are spawned on the same machine of the cluster.

Considering that reducing cluster memory usage can be achieved for a single function, we began to explore the potential of leveraging this feature in the context of serverless workflows.

### B. Memory Sharing Potential of FaaS Workflows

With the development of serverless computing, when migrating existing applications to FaaS platforms, developers typically break them down into multiple functions to form complex workflows [17], [18], [19] and FaaS platforms also provide such orchestration mechanisms [20], [21]. Workflows are usually represented as Directed Acyclic Graphs (DAGs) with each function serving as a vertex in the graph. To evaluate the memory sharing

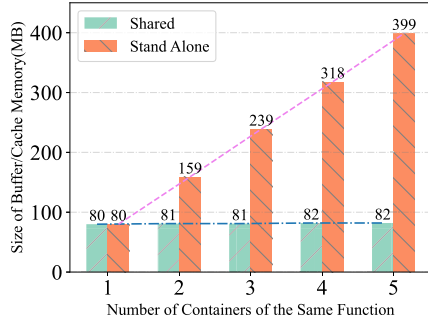


Fig. 3. Buffer/Cache memory usage after spawning work-flow function containers.

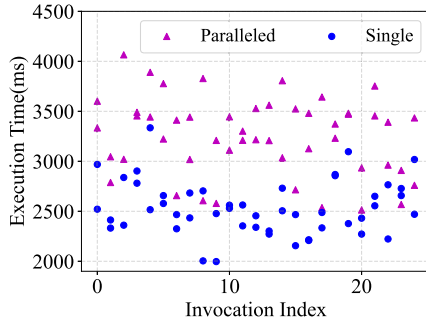


Fig. 4. Parallel invoking function would perform worse than invoking one at a time.

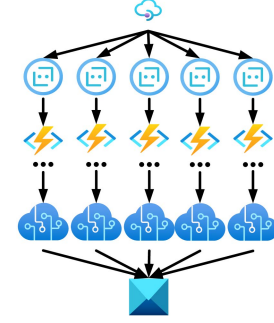


Fig. 5. Structure of a DAG.

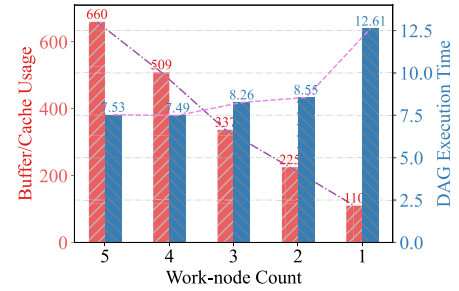


Fig. 6. Change of Cache memory usage and execution time.

potential, we utilized ServerlessBench [22] and conducted experiments on OpenWhisk. During this process, we made several discoveries.

*First, functions within the same workflow often share common underlying images.* For example, in an image processing workflow, multiple functions may use the same Python runtime layer at the bottom, with unique layers on top for each function. A detailed decomposition is shown in Fig. 1. This pattern is also observed in other workflows, such as web applications and IoT processing applications whose functions usually use same run-times and similar libraries. [22] *Additionally, parallelism is common in serverless workflows.* Heavy computation tasks are usually split into parallel sequences. Parallel paths in a workflow often consist of similar or the same function chains. This leads to cases similar to placing the same functions' containers together, as shown in Fig. 2. *Finally, co-placing containers of the same workflow does save memory.* We then selected a very simple workflow with five functions and conducted another comparison. This workflow's function containers are placed into same (Shared) and different (Stand Alone) machines during the two isolated tests. Results in Fig. 3 illustrates the significant reduced memory consumption of the cluster when containers are placed together.

These discoveries reveal the great potential of function memory sharing. Leveraging the feature of OverlayFS would save a significant amount of memory resources for the cluster while processing workflows. However, we encountered a non-negligible drawback when testing a more complicated DAG.

### C. Execution Time Deterioration

During further tests, we evaluated a workflow with the structure shown in Fig. 5. It's a way more complex and larger workflow DAG with dozens of functions. We placed the containers of the five sequences into one or five work-nodes and recorded related stats. As shown in Fig. 6, a large amount of cache memory was saved when placing the containers together. However, we also recorded the workflow's execution time. The results in Fig. 6 show that the execution time of placing all containers together was significantly larger than placing them independently. This may be because of running paralleled functions together would impose too much pressure on the work-node, leading to the deterioration of the workflow's execution time.

To test our assumption, we invoked a simple but CPU-sensitive function *ImageFilter* in two different ways: making one invocation at a time or making five invocations arrive simultaneously on the same machine. As shown in Fig. 4, it is evident that when one machine handles one invocation at a time, the execution time is observably lower than when handling multiple paralleled invocations simultaneously. We then placed *ImageFilter*'s container into machines with increasing resource occupancy for testing. The system pressure is set using the Linux command *STRESS*. No difference was observed as memory occupancy increases. But Fig. 7 illustrates that as the machine's CPU occupancy rate increases, the function's execution time also increases, reaching exponential growth after a certain point. We call this the *cut-off point*. This finding supports our opinion. Fig. 7 also shows the statistics of four other functions with

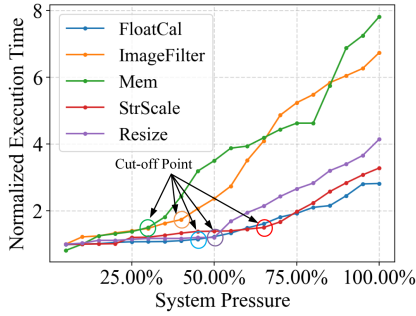


Fig. 7. Function execution time would increase as system pressure grows.

different memory and CPU requirements. Their results all indicate that high CPU pressure can degrade function performance, regardless of function characteristics.

Focusing solely on sharing can lead to sub-optimal execution results. Reducing memory consumption can be simply achieved by placing all containers of the DAG on the same machine, but the workflow's execution would cost much more time if we do not pay attention to CPU pressure. Serverless workflows usually have SLO (Service Level Objective) constraints which require acceptable execution time, thus such deterioration would not be acceptable.

Back to Fig. 6, we changed the number of work-nodes from 1 to 5 and tried to evenly place the five sequences. It's clear that placing containers in two work-nodes would not increase the execution time too much, but a great deal of memory has already been saved. And for each curve in Fig. 7, the execution time increases slightly until the machine's system pressure reaches the cut-off point.

Therefore, we argue that there should be a way to find a balance point between memory consumption reduction and workflow SLO guaranteeing. Placing function containers accordingly would save memory while satisfying SLO. That is what this paper will work for. Aiming at that, we design and implement two heuristic function placement algorithms, namely MDP and GSP, to tackle function placement within serverless clusters. They leverage the sharing potential of function image layers of a DAG to perform function container placement. MDP focus on static workflows and GSP handles dynamic ones.

It is noteworthy that the preceding discussion exploits the features of Docker containers, enabling the achievement of layer-wise memory sharing. Other serverless platforms designed for production, such as AWS Lambda [2] and Azure Function [3], possess their own dedicated function execution sandbox environments, such as microVM or Azure Function Runtime. These platforms do not rely on OverlayFS as the underlying file system, and consequently, they are not the subject of experimentation or discussion within the scope of this paper.

#### D. Safety Discussion

Here, we would like to first address the safety concerns associated with layer-wise memory sharing. Our perspective is that sharing page cache memory among Docker containers is

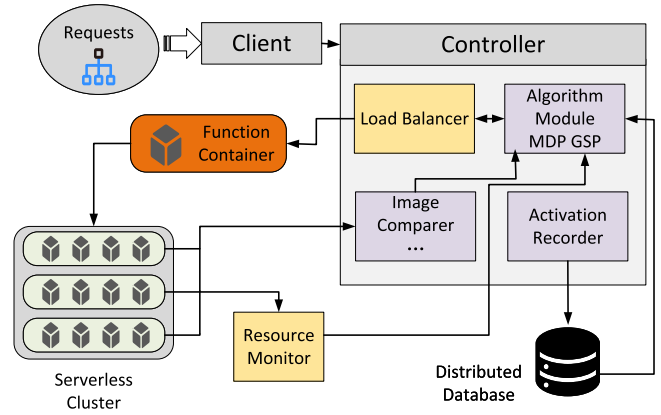


Fig. 8. System overview.

unlikely to pose significant security risks or compromise user privacy. To begin with, the shared layers are typically read-only and primarily reserved for constructing essential runtime environments. As a result, the shared memories do not encompass memory areas critical to program execution, such as the stack or heap.

Furthermore, these shared pages commonly consist of foundational components, including libraries and system binaries, which do not contain sensitive information. Additionally, the unique layers containing sensitive information for each container are generally not shared, providing an additional layer of isolation. While it is true that shared page caches involve a fundamental level of sharing, the limited nature of this sharing, coupled with its read-only characteristics and focus on non-sensitive components, helps mitigate potential security issues.

Last but not least, the sharing we intend to leverage is within serverless workflows belonging to the same users. This further reduces the risk of privacy violations. Overall, in scenarios where shared pages are carefully managed and restricted to non-sensitive elements, the security impact of page cache memory sharing is generally minimal.

### III. SYSTEM OVERVIEW

MDP and GSP are function placement algorithms designed to consider layer-wise memory sharing between function containers while ensuring adherence to workflow SLO requirements. These algorithms are implemented on the serverless platform OpenWhisk [13], a versatile open-source project widely employed not only for commercial purposes in IBM Cloud Functions but also in numerous scientific research endeavors.

In Fig. 8, the architecture of our system is illustrated, where the light purple blocks represent the modules we added to the platform. The Controller, central management entity of OpenWhisk responsible for request processing, scheduling, and resource management, plays a crucial role in our design. Our modifications include two main components.

The first component is a modified activation recorder, enhancing the Controller's ability to record each function's execution time and the system's resource pressure during execution. As



discussed earlier and shown in Fig. 7, function execution time is influenced by system pressure, and the *Cut-off Point* is of significant importance. This modified recorder generates execution curves, akin to those in Fig. 7. The recorded data is stored in a new table of OpenWhisk's database for MDP and GSP to utilize. Recorder is also responsible for calculating and locating the *Cut-off Point*. For a curve with a sudden change in slope, as shown in Fig. 7, the Cut-off Point is the point where the slope abruptly changes. If there are no slope transition points in the curve, two scenarios below are considered. First, if the function execution time steadily increases with the growth of system pressure, the point at which the execution time is **N** times of the minimum execution time is selected. Second, if the function execution time hardly changes with the increase in system pressure, the point at which the system pressure is **M**% is chosen. **M**% would be rather large like 80% or 90%. Here, both **N** and **M** are configurable values. They serve the purpose of preventing excessive execution time and excessive system pressure. It is worth noting that even for simple functions in our experiments, there are still obvious cut-off points. The purpose of these two strategies above is to enhance the robustness of the system in specific scenarios.

The second component is the Algorithm module, which implements MDP and GSP. Detailed discussions of these two algorithms will be presented in Section V. This module takes input from various sources, calculates the optimal function placement decision, and communicates with the Load Balancer to execute the actual function placement. There are also modules with small modification, such as the Image Comparer, serving as auxiliary components. All these modules are written in Scala as OpenWhisk originally use and their details will be discussed in Section VI.

Our system operates with the following pattern: The Activation Recorder records the execution time of serverless workflow functions and the associated system pressure. It generates execution curves and locates the cut-off point. When a workflow request arrives from the client, the Algorithm Module obtains input data from the database, the cluster's Resource Monitor, and other auxiliary modules. It then executes algorithm MDP for static workflows or GSP for dynamic workflows. The computed placement result is then sent to the load balancer for container placement. Through the integration of these components, our system collectively performs function placement to maximize memory sharing while guaranteeing the workflow's SLO requirements.

With system overview presented here, subsequent paragraphs will delve into mathematical modeling, detailed algorithms, and the actual implementation.

#### IV. MODELING

##### A. Workflow Model

A serverless workflow contains multiple functions that perform logical computations for an application. Its structure determines the overall characteristic of the application. We first provide a model for the workflow.

Notations	Definition
$G = (V, E)$	DAG for the function workflow
$F$	Function Set of the function workflow
$W$	Work-node set of the cluster
$G_{sub} = (V_{sub}, E_{sub})$	a sub-graph of the DAG $G$ .
$v_i$	A vertex in the DAG.
$s_m$	The $m$ th step of the DAG.
$f_j$	A function of the workflow, mapped to one or more vertices.
$w_k$	A work-node of the serverless cluster.
$o_k$	The resource occupancy of $w_k$ .
$T$	The overall execution time of DAG $G$ .
$o_k^j$	Work-node's resource occupancy at cut-off point of the Pressure-Time curve.
$\xi(o_k, f_j)$	Function $f_j$ 's execution time when placed in $w_k$ .
$\Xi$	All $\xi$ curves of the workflow.
$PSS(f_m, f_n)$	The Potential Sharing Score (PSS) between functions $f_m$ and $f_n$ .
$\delta(v_i, w_k)$	The sharing score when $f_j$ is placed on $w_k$ .
$x_i(w_k)$	Whether function at $v_i$ is placed in $w_k$ .
$G_{sub}^{upper}$	The upper benefit bound of a sub-graph.
$G_{sub}^{lower}$	The lower benefit bound of a sub-graph.
$P_{crucial}$	The crucial path of a workflow.
$G_{dynamic}$	The dynamic part of a workflow DAG.
$G_{static}$	The static part of a workflow DAG.
$\theta(s_m)$	The weight of a step in GSP.
$T_{s_m}$	The Sub-SLO of a step.

Workflows are most commonly described as Directed Acyclic Graphs (DAGs). We use  $G = (V, E)$  to characterize a workflow, where the vertex set  $V = \{v_1, v_2, \dots, v_n\}$  represents the vertices within the graph, and the edge set  $E = \{\widehat{v_i v_j} | 1 \leq i \neq j \leq n\}$  represents the edges connecting the vertices. The symbol  $\widehat{v_i v_j}$  represents an edge from vertex  $v_i$  to vertex  $v_j$ , indicating that the function at  $v_j$  would be invoked after the execution of the function at  $v_i$ . Note that in serverless DAGs, different vertices may represent the same function located differently in the graph due to parallelism. We use the function set  $F = f_1, f_2, \dots, f_m$  to represent the functions included in the workflow.

We next model the *Steps* of workflow DAGs. As in many prior works [6], [11], [23], DAGs are often orchestrated as a combination of multiple steps. A step in a DAG consists of nodes that have the same depth or level in the graph. Steps in a DAG are often used to organize and parallelize the execution of tasks. Functions within the same step can be executed concurrently since they have no direct dependencies on each other. For example, all the successor nodes of a fan-out node belong to the same step, and they run concurrently. In this paper, symbol  $v_i^s$  represents the step of  $v_i$ , where  $v_i^s > 0$ . For convenience, we use  $G_{sub} = (V_{sub}, E_{sub})$  to represent a sub-graph of the DAG. As nodes of the same step have no direct edge connections to each other, a step can be denoted as a sub graph with no edges. The  $m$ th step could be described as  $s_m = (V_{sub} = \{v_i | v_i^s = m\}, E_{sub} = \emptyset)$ .

##### B. Function Execution Model

We now have a model of the workflow. As each vertex in the graph has a related function, we must discuss the characteristics of these functions. Additionally, we need to consider the features

of work-nodes (machines) in the cluster where these functions are deployed.

We model the cluster in which serverless functions are deployed as  $W = \{w_1, w_2, \dots, w_k\}$ , where  $w_k$  represents a work-node of the cluster. Experiments in Section II indicated that work-node's CPU occupancy plays an important role in affecting functions execution time while no difference was observed as memory occupancy increases. So we model  $w_k$ 's *System Pressure* as its *CPU Pressure*, denoted as

$$o_k = 1 - \epsilon \frac{w_k^{ac}}{w_k^c}, \quad (1)$$

$\epsilon$  is a parameter for adjustment.  $w_k^c$  and  $w_k^{ac}$  are the overall and available CPU of a work-node, which can be obtained by directly monitoring work-nodes' status.

FaaS platforms typically allow users to configure the memory size for their functions, and the CPU resources allocated are directly affected by it [2], [3]. This strategy aligns with the approach adopted by OpenWhisk [24]. We denote the allocated memory and CPU requirement of function  $f_j$  as  $f_j^m$  and  $f_j^c$ , respectively.  $f_j^c$  is in positive correlation with  $f_j^m$ , i.e.,  $f_j^c = \alpha f_j^m$ .  $\alpha$  is a parameter determined by the ratio of the platform's CPU to memory [2], [3]. With the modeled resource configuration, we proceed to model the resource occupancy of functions. Functions can be either compute-bound or memory-bound, thus having different memory-CPU occupancy ratios. Given a configured memory size, we denote the CPU occupancy of function  $f_j$  as

$$f_j^{c-occup} = \gamma \alpha f_j^m. \quad (2)$$

Each function has its own  $\gamma$  value calculated from historical records, determining its memory-CPU occupancy fraction.

Apart from resource requirements, another important metric to consider is the execution time of functions. Let the execution time of  $f_j$  be  $f_j^t$ . As shown in Fig. 7, with a fixed resource requirement, function's execution time  $f_j^t$  first changes slightly as the pressure on the work-node grows. But when the resource occupancy reaches a certain extent, the execution time would grow rapidly. We use  $o_k^g$  to denote the cut-off point of resource occupancy. Let  $t_{j,k}^\gamma$  be the time before  $o_k^g$  and  $t_{j,k}^\tau$  be the deteriorated time. So, when  $f_j$  is placed on  $w_k$ , we define the function execution time  $f_{j,k}^t = \xi(o_k, f_j)$  as

$$\xi(o_k, f_j) = t_{j,k}^\gamma [o_k \leq o_k^g] + t_{j,k}^\tau [o_k > o_k^g]. \quad (3)$$

We use  $\Xi$  to represent all  $\xi$  curves of the workflow. Note that function execution time is always recorded in FaaS platforms as it's the metric cloud vendors leverage to charge their users. By recording the system occupancy simultaneously, the data needed for  $f_j^t$  would be available.

We next model the execution time of the workflow. The overall execution time of  $G$ , denoted as  $T$ , depends on the path with the longest running time in  $G$ . Let  $P = \{p_1, p_2, \dots, p_p\}$  denote the collections of vertices that travel from the starting point to the end in the DAG.  $T$  is then formulated as

$$T = \max_{p \in P} \sum_{v_i \in p} f_{j,i}^t, f_{j,i} \leftarrow v_i. \quad (4)$$

This path will be called the crucial path, denoted as  $P_{crucial}$ .

### C. Function Sharing Model

As mentioned before, functions belonging to the same workflow have the ability to share cache memory usage. In this paper, we use  $\delta(v_i, w_k)$  to denote the sharing of  $v_i$  when its function container is placed in  $w_k$ .

The basic sharing among function containers is introduced by page cache sharing. *Pages* are *disk files* mapped to memory, and disk files are determined by *container layers*. For two coexisting containers, page cache could be shared if they access the same layer [16]. These layers typically consist of runtime-specific libraries and system binaries, and datasets may also be included as layer files if they are part of the image. However, datasets introduced by function inputs would not be taken into account. We cannot exactly calculate how much page cache two containers share, as the files they would access are determined by functions themselves. But an approximate sharing value could be obtained by measuring the sizes of two images' common layers. We use  $l$  to represent a layer,  $l^s$  to represent the size of a layer, and  $f_j^l$  to represent all the layers of  $f_j$ . Consequently, the *Potential Sharing Score (PSS)* between two functions is

$$PSS(f_m, f_n) = \frac{\sum_{l \in f_m^l \cap f_n^l} l^s}{\sum_{l \in f_m^l \cup f_n^l} l^s}. \quad (5)$$

Also, the sharing score of putting  $f_j$ 's container in  $w_k$  is determined by the containers already existing in  $w_k$ . When  $f_j$  is placed on  $w_k$ , we denote its sharing score as

$$\delta(v_i, w_k) = \sum_{l \in f_j^l, f_j \leftarrow v_i} l^s \text{ where } l \in f_j^l \ \& \ l \in \bigcup_{f_m \rightarrow w_k} f_m^l. \quad (6)$$

Furthermore, we use a binary variable to indicate whether function  $f_j$  in  $v_i$  is placed in  $w_k$

$$x_i(w_k) = \begin{cases} 0, & f_j^t \text{'s container is not placed in } w_k \\ 1, & f_j^t \text{'s container is placed in } w_k \end{cases}. \quad (7)$$

As in one execution, each container of a vertex can only be placed once, we have the constraint

$$\sum_{w_k \in W} x_i(w_k) = 1, \forall v_i \in V. \quad (8)$$

## V. PROBLEM FORMULATION AND SOLUTIONS

Our objective is to maximize the memory sharing among function containers while maintaining a balanced and SLO-aware state for the workflow. The Service Level Objective (SLO) for the workflow is denoted as  $T_{SLO}$ , representing the maximum allowed execution time for the entire workflow.  $T_{SLO}$  represents a user-defined workflow execution time limit. We assume both the user and the platform are rational; the user would not set an excessively high  $T_{SLO}$  to avoid unnecessary costs, and the platform would not allow an overly low  $T_{SLO}$  to maintain user satisfaction and platform credibility.

$$T = \max_{p \in P} \sum_{v_i \in p} f_j^t < T_{SLO}. \quad (9)$$

**Algorithm 1:** Sharing-Aware Container Placement.

---

**Input:** DAG  $G = \{V, E\}$  where vertex  $v_i \in V$ ;  
 Functions mapped to  $v_i, f_j \rightarrow v_i$ ; Function  
 execution time curve  $\xi(o_k, f_j)$ ; Info of all  
 work-nodes,  $W$ ; Workflow SLO,  $T_{SLO}$   
**Output:** Work-nodes chosen to place containers of  
 $f_j \in V$

- 1 Calculate all placement options  $X_{all} \leftarrow \{X|x_i(w_k)\}$ ;
- 2 Filter out options that doesn't fit topological need,  
 $X_{topo} \leftarrow X_{all} - \{X|x \rightarrow \text{untopological}\}$ ;
- 3 **for each**  $X \in X_{topo}$  **do**
- 4   Calculate DAG execution time  $T_X$ ;
- 5   **if**  $T_X \leq T_{SLO}$  **then**
- 6     Store  $X$ ;
- 7     Calculate memory saved due to sharing  $S_X$ ;
- 8   **end**
- 9 **end**
- 10  $X_{res} \leftarrow X$  who has  $\max(S_X)$ ;

---

To achieve this, we need information about the workflow's Directed Acyclic Graph (DAG) for placement calculations. However, in reality, the full DAG might not always be accessible due to various programming patterns or privacy concerns of users. In some cases, workflows are orchestrated using triggers or in-function invocations, making the DAG structure dynamic and responsive to real-time requests or different inputs. Therefore, we present two distinct problems and their solutions to address both static and dynamic DAG scenarios.

#### A. Container Placement With Full Knowledge of DAG

The goal of the *Static Shared Function Placement (SSFP)* is to maximize the sharing ( $S$ ) of function containers while considering the DAG's overall topology structure, the features of functions in each DAG vertex, and a set of cluster work-nodes ( $W$ ). The main constraint is to ensure that the workflow's SLO is guaranteed. We formulate the problem as:

**Problem 1: (SSFP)** Given  $G, W, F$  and  $\Xi$ , calculate (4) and (6), subject to constraints (7), (8) and (9), to maximize  $S = \sum_{v_i \in G} \delta(v_i, w_k) \cdot x_i(w_k)$

1) *A Straightforward Solution:* An intuitive approach to finding the optimal work-node to place all containers ( $v_i$ 's) is to use Algorithm 1.

Algorithm 1 first finds all possible placement options and ensures that they adhere to the topological order (successor nodes are placed earlier than predecessor nodes). It then calculates the execution time of each option and filters out options that do not meet the SLO requirement (Line 3–9). For each placement, Algorithm 1 calculates the sum of sharing for the workflow and selects the option with the highest sharing. Although this idea is straightforward, the following Theorem 1 shows that the SSFP is NP-hard, which makes it impractical.

**Theorem 1:** Problem SSFP is NP-Hard.

**Proof:** We prove this theorem by mapping the problem SSFP to a multi-knapsack problem where item values differentiate in different packages. Given a multi-knapsack problem (MKP)

$$\begin{cases} I = \{i_1, i_2, \dots, i_n\}, \text{ each has a positive value } u_i. \\ P = \{p_1, p_2, \dots, p_m\}, \text{ each has a capacity } c_j \\ \text{Each item has a weight } w_i \\ \sum_{j=1}^m w_i \cdot x_{ij} \leq c_j, x_{ij} \in \{0, 1\} \\ \text{Solve } \max V = \sum_{i=1}^n u_i \cdot x_{ij} \end{cases} \quad (10)$$

$I$  is the set of items, each item has a value  $u_i$ .  $P$  is the set of backpacks and  $c_j$  represents each backpack's capacity.  $x_{ij}$  stands for whether an item is placed in a backpack. Back to SSFP, regard each  $w_k$  as a knapsack with capacity  $1 - o_k$ . Each  $f_j$  is an item to be put into the backpack. The value of the item  $s_j$  is determined by not only the item itself, but also by the chosen backpack's capacity  $o_k$  and item already in it. And the goal of the problem SSFP is to get the max score  $S$ . We map it to Problem (10) with  $\{v_i \leftarrow i_i, w_k \leftarrow p_j, s_i \leftarrow u_i, o_j \leftarrow c_j, S \leftarrow S\}$ . Generally speaking, in our model, problem SSFP is a multi-knapsack problem where item values differentiate according to backpack capacity and items already in it. So if there exists an algorithm that solves our problem, it also solves the corresponding multi-backpack problem. Owing to the NP-Hardness of multi-knapsack problem, SSFP must be NP-Hard as well.  $s_j \leftarrow S(o_k, f_m \in w_k, G)$ . And multi-knapsack problem is known to be NP-Hard [14].

2) *Heuristic Solution With Full Knowledge of DAG:* In this section, we aim to reduce the computational complexity of SSFP by leveraging heuristic insights derived from the DAG structure and function execution variations. We propose a practical algorithm called Merged DAG-Partition (MDP) to efficiently choose the appropriate work-nodes for placing workflow's function containers.

The sharing among function containers arises from the use of the same layers in their images. Our first key insight is that *Insight (1) functions belonging to the same step have a great potential for sharing and performance improvement*. This concept was explained in Section II. Another important insight is that *Insight (2) sharing can happen among sequencing functions of a DAG with low CPU pressure increase*. After invocation, containers are usually retained in the work-nodes due to the keep-alive policy. Keep-alive containers have minimal CPU consumption but still occupy memory. Hence, successor nodes can directly utilize the cached pages, resulting in no deterioration since they often have similar underlying dependencies.

Based on two these insights, we introduce the heuristic algorithm called **Merged DAG-Partition (MDP)** to address problem SSFP. MDP partitions the entire DAG into several sub-graphs, where functions of the same sub-graph are placed in the same work-node (Algorithm 2). Mechanism of MDP is given below and Fig. 9 provides an example of it.

*Mechanism One, the partition of DAG is achieved by merging vertices of DAG vertically and horizontally:* The ultimate goal of MDP is to place function containers into different work-nodes. Vertices of functions within a work-node forms a sub-graph of the original DAG. Based on *Insight (1)* above MDP would horizontally merge vertices of the same step/stage of the DAG. On the same basis, vertical merge is done based on *Insight (2)*. Horizontally merged functions belong to the same stage of the workflow, so they would be executed concurrently, posing



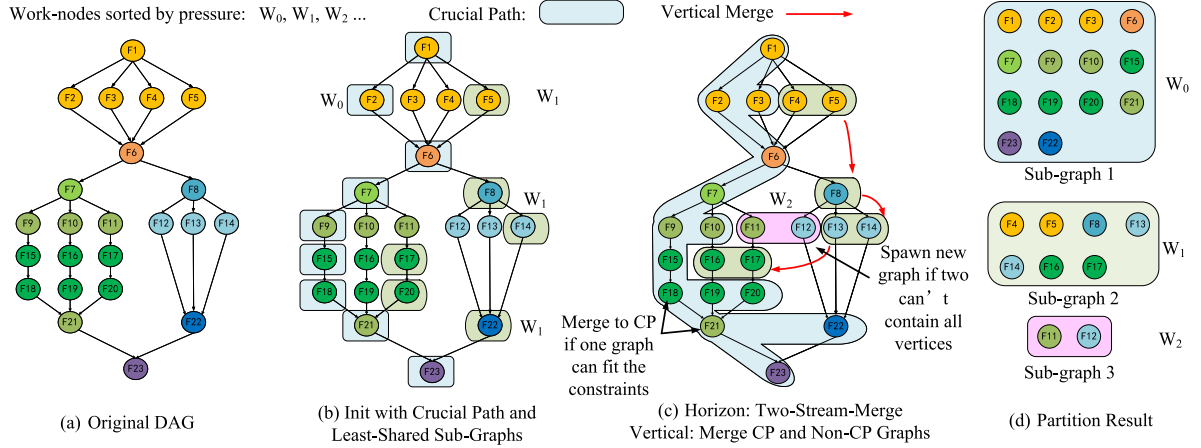


Fig. 9. Example of merged DAG partition (MDP).

**Algorithm 2: Merged DAG Partition.**


---

**Input:** DAG  $G = \{V, E\}$  where vertex  $v_i \in V$ ;  
 Functions mapped to  $v_i, f_j \rightarrow v_i$ ; Function  
 execution time curve  $\xi(o_k, f_j)$ ; Info of all  
 work-nodes,  $W$ ; Workflow SLO,  $T_{SLO}$

**Output:** Work-nodes chosen to place containers of  
 $f_j \in F$

- 1 Select several work-nodes with low pressure  $W_{low}$  from  $W$ ;
- 2 Find the crucial path  $P_{crucial}$  for  $G$ , use its vertices as initial sub-graphs;
- 3 Topologically partition workflow  $G$  into steps,  $S = \{s_1, s_2, \dots, s_m, \dots, s_n\}$ ;
- 4 **for** each step  $s_m \in S$  **do**
- 5   Use crucial path function vertex  $v_{CP}^m$  as base, sort function vertices  $v_i^m$  by (5);
- 6   Get Least Shared  $v_{LS}^m$  with lowest PSS;
- 7   Init two sub-graphs  $G_{CP}^m \leftarrow v_{CP}^m$  and  $G_{LS}^m \leftarrow v_{LS}^m$ ;
- 8   Init  $G_{CP}^m, G_{LS}^m$  pressure with  $W_{low,0}, W_{low,1}$ ;
- 9   **while** Satisfy  $G_{sub}^{upper}, G_{sub}^{lower}$  AND  $T_{P_{crucial}} \leq T_{SLO}$  **do**
- 10    **if** only  $G_{CP}^m$  and  $G_{LS}^m$  exists **then**
- 11    |  $G_{CP}^m$  merge  $G_{LS}^m$ ;
- 12    **end**
- 13    Calculate sharing score with (5);
- 14     $G_{CP}^m$  merge  $v_i \leftarrow highestShare(v_{CP}^m)$ ;
- 15     $G_{least}^m$  merge  $v_i \leftarrow highestShare(v_{least}^m)$ ;
- 16   **end**
- 17   **if** has  $v_i$  remaining AND has  $W$  remaining **then**
- 18    | Init new sub-graph  $G_{extra}^m$  and perform TSM;
- 19   **end**
- 20 **end**
- 21 **for**  $G_{CP}^m, G_{LS}^m$  and  $G_{extra}^m$  of each step **do**
- 22   | Merge vertically;
- 23 **end**
- 24 Place functions of the same sub-Graph into the same  $W_{low}$ ;

---

pressure to the work-node. Vertically merged functions belong to sequencing stages of the workflow, they won't increase pressure to the work-node at the same time but their execution time would be added up. Accordingly, we have Mechanism Two and Three.

*Mechanism Two, merge has limits of CPU pressure and memory sharing benefits:* As we experimented before, horizontally adding a vertex into a sub-graph can increase the benefit of memory sharing but also introduce more CPU pressure. The sharing benefit and pressure can be obtained using (6) and  $f_j^c$ . Certainly, it's possible that after one merging, it may bring about too little sharing benefit but too much pressure. To prevent such useless or harmful merge, we set an upper limit for resource pressure and a lower bound for memory sharing benefit for each sub-graph. After horizontally adding a vertex, the resource occupancy cannot exceed the resource limit, and the benefit from sharing cannot be lower than the benefit bound. MDP sets the upper bound using the function in the sub-graph who has the smallest execution time at its cut-off point. The resource pressure at this point is used, thus upper bound is calculated as

$$G_{sub}^{upper} = \min_{f_j \in G_{sub}} o_k^g \leftarrow \xi(o_k, f_j). \quad (11)$$

The lower bound is a user-defined value indicating the minimum percentage of shared layers. The lower it is, the more tolerant MDP would presented as.

$$G_{sub}^{lower}. \quad (12)$$

As for vertical merge, only sharing limit should be considered because CPU pressure would not add up.

*Mechanism Three, MDP guarantees SLO by controlling the execution time of the crucial path:* Recall that we previously modeled the workflow's execution time as determined by its crucial path  $P_{crucial}$ . We use the work-nodes' average occupancy to calculate each function's execution time using (3). These calculated execution times are used to obtain  $P_{crucial}$ , which contains functions with the longest execution time for each step. Summed execution time of crucial-path functions is the execution time of the workflow. While performing horizontal merges MDP would guarantee that 1) the execution time of none crucial-path functions won't exceed crucial-path function and 2)  $T_{P_{crucial}} \leq T_{SLO}$ . Under such circumstances, constraint (9) is satisfied.



*Mechanism Four*, MDP performs horizontal merging using a Two-Stream Merge algorithm and vertical merging by distinguishing crucial-path sub-graphs from non-crucial-path ones: Having discussed the principles of merging in three mechanisms above, Mechanism Four would elaborate how MDP carries out its two-dimensional merge. MDP performs horizontal merge using a Two-Stream Merge (TSM) algorithm. TSM sorts the functions of a step in descending order by comparing their sharing scores with the  $P_{crucial}$  function using (5) (Line 5). Then, it selects the  $P_{crucial}$  function ( $f_{CP}$ ) and the function with the least sharing score ( $f_{LS}$ ) with  $f_{CP}$  as two starting points of the step (Line 6-8). This aims at reducing merging complexity and increasing sharing score compared to randomly selecting starting points. The two starting sub-graphs then iteratively merges functions that have the highest sharing scores with themselves while satisfying the constraints in Mechanism Two. If, in the end, only two sub-graphs remain, MDP merges them as long as the constraints are met. If two graphs cannot contain all vertices, a new sub-graph is added, and the process continues (Line 9-19). But if the number of sub-graphs reaches the number of work-nodes, no more sub-graphs should be created and remaining vertices should be added to existing sub-graphs. The number of work-nodes is obtained from the serverless platform, which is OpenWhisk in our case. After horizontal merge, vertical merge is done. MDP first vertically merges the crucial path sub-graphs together. They serve the purpose of restricting SLO. The second merge merges non-crucial-path sub-graphs to obtain larger sub-graphs in the vertical level (Line 21-23).

Fig. 9 illustrates an example of a MDP. The process begins with a DAG and work-nodes, where work-nodes are initially sorted based on pressure. (In Fig. 9(a),  $W_0, W_1$  are work-nodes) For partitioning, MDP first identifies crucial path functions for the DAG using Mechanism Two. (In Fig. 9(b), the crucial path functions are located and so are the least shared functions.) Subsequently, merge operations are conducted under Mechanism Four, and sharing and pressure computations follow Mechanism Two (Fig. 9(c)) For example, F3 and F4 merged into two separate sub-graphs instead of just the crucial path because adding them together would exceed the CPU pressure limited by Mechanism Two. But F18, F19 and F20 can all be merged together because their pressure together is not that high. Finally, the vertical merge is executed using Mechanism Four (Fig. 9(c) and (d)). Ultimately, the functions within each sub-graph are placed in the same pre-selected work-node.

For a DAG with  $N$  steps, the time complexity is  $O(|V| \log \frac{|V|}{N} + W)$ , where  $|V|$  is the number of vertices in the DAG.  $W$  is the number of work-nodes. This complexity arises from sorting work-nodes, crucial path calculation, and horizontal merging using the Two-Stream Merge algorithm where parsing vertices and sorting  $PSS$  is done in every step. The space complexity is  $O(|V|)$ , accounting for storing the crucial path, sub-graphs, and additional structures for calculations.

The approximation ratio of the algorithm MDP for SSFP is 2 and proof is given using Theorem 2 below.

*Theorem 2:* The approximation ratio of the algorithm MDP for SSFP is 2.

*Proof:* We provide the sketch of the proof. Horizontally, MDP merges vertices into the same sub-graph by sharing score value. Same to the mapping in Theorem 1, we can also map horizontal sub-graph merge to a MKP. So within each step, MDP solves the MKP in a value-based pattern. It has been proved in [25] that using such strategy has a gives a 2-approximation rate. As Insight 2 indicate that vertical lead to minor pressure influence, vertical impact could be ignored. We argue that MDP has a approximation ratio of 2 for SSFP. Due to page limitation, detailed explanations are omitted.

### B. Container Placement for Dynamic DAGs

Next, we address the solution for dynamic scenarios where the DAG's structure can change depending on inputs or configurations [26]. Our focus is on workflows characterized by a partially static structure, where the uncertainty lies in the potential invocation of several functions. Specifically, the number of parallel sequences exhibits dynamism, and the invocation of multiple functions is determined in real-time. It's essential to note that these dynamic aspects should not involve cycles or loops, as our target domain is directed acyclic graphs (DAGs) as prior works like [6], [11], [26] In this case, we have only partial knowledge of the DAG structure. The other parts, for example whether several branches would be invoked or whether function  $F_A$  or  $F_B$  would be invoked is determined only during invocation. Let  $G_{dynamic}$  denote the dynamic part of DAG, and the known part is given by

$$G_{known} = G - G_{dynamic}. \quad (13)$$

Therefore, the problem *Dynamic Shared Function Placement (DSFP)* is to maximize the sharing  $S$  of function containers, given information about cluster work-nodes, partial topology structure of the DAG, and features of functions. The workflow's SLO should be met as a constraint. In summary, we have the following optimization problem:

*Problem 2: (DSFP)* Given  $G_{known}$ ,  $W$ ,  $F$  and  $\Xi$ .  $G_{dynamic}$  is accessible during placement procedure. Calculate (4) and (6). Try maximize  $S = \sum_{v_i \in G} \delta(v_i, w_k) \cdot x_i(w_k)$ , subject to constraints (7), (8) and (9).

*Theorem 3:* Problem DSFP is NP-Hard.

*Proof.* DSFP is a more complicated variation of SSFP with additional constraints. It is essentially an online SSFP with the arrival of dynamic functions known only during the placement procedure. We can map DSFP to a MKP with items information known only during the procedure of putting them into packages. Given a modified multi-knapsack problem (MKP)

$$\left\{ \begin{array}{l} I = \{B_1, B_2, \dots, B_n\} \\ B_i = \{i_1, i_2, \dots, i_n\}, \text{ each } i \text{ has a positive value } u_i. \\ P = \{p_1, p_2, \dots, p_m\}, \text{ each has a capacity } c_j \\ \text{Each item has a weight } w_i \\ \sum_{j=1}^m w_i \cdot x_{ij} \leq c_j, x_{ij} \in \{0, 1\} \\ \text{Solve } \max V = \sum_{i=1}^n u_i \cdot x_{ij} \end{array} \right. \quad (14)$$

**Algorithm 3: Greedy Step Placement.**


---

**Input:** DAG  $G = \{V, E\}$  where  $V$  and  $E$  are dynamic;  
 Functions mapped to  $v_i, f_j \rightarrow v_i$ ; Function  
 execution time curve  $\xi(o_k, f_j)$ ; Info of all  
 work-nodes,  $W$ ; Workflow SLO,  $T_{SLO}$

**Output:** Work-nodes chosen to place containers of  
 $f_j \in V$

- 1 Topologically partition workflow  $G$  into steps,  
 $S = \{s_1, s_2, \dots, s_m, \dots, s_n\}$ ;
- 2 Calculate sub-SLO  $T_{s_m}$  for each each  $s_m$  using (17);
- 3 Monitor function requests;
- 4 **if** step requests for  $s_m$  detected **then**
- 5    $F \leftarrow f_j \Leftrightarrow s_m$ ;
- 6   Get sub-SLO  $T_{s_m}$ ;
- 7    $w_{select} \leftarrow w_k \in W$  where  $w_k$  has lowest  $o_k$ ;
- 8   Try place  $f_j \in F$  in one node;
- 9   Get execution time from  $\xi(o_k, f_j)$ ;
- 10   **while**  $T_{max} > T_{s_m}$  **do**
- 11     AddAndBalance( $w_k, W, s_m$ );
- 12     Break if reaches limit;
- 13   **end**
- 14   Record and Apply  $X \leftarrow \{x_i | v_i \in s_m\}$ ;
- 15 **end**
- 16 **Function** AddAndBalance( $W_{old}, W, s_m$ ):
- 17    $F \leftarrow \{f \in \{W_{old}\}, f \Leftrightarrow v_i \text{ where } v_i \in s_m\}$ ;
- 18   Select a new work-node  $w_k$  from  $W$ ;
- 19   **if** inBatch **then**
- 20      $W_{new} = W_{old} + w_k$ ;
- 21     for all  $f_j \in F$ : Evenly place  $f_j$  into  $W_{new}$ ;
- 22   **end**
- 23 **end**

---

$I$  is the set of *items batches*, each batch has several items which are known only when actually handling the batch. The rest definitions are the same as *SSFP*. Regard each  $w_k$  as a knapsack with capacity  $1 - o_k$ . Each  $f_j$  is an item to be put into the backpack. The value of the item is  $s_j$ . The goal of the problem *DSFP* is to get the max score  $S$ , the placement should be executed with batches arriving. We map it to Problem (14) with  $\{v_i \leftarrow i_i, w_k \leftarrow p_j, s_i \leftarrow u_i, o_j \leftarrow c_j, S \leftarrow S\}$ . So if there exists an algorithm that solves our problem, it also solves the corresponding dynamic multi-backpack problem. Owing to the NP-Hardness of multi-knapsack problem [14], *DSFP* must be NP-Hard as well. Therefore, *DSFP* is also NP-Hard.

To tackle *DSFP*, we need an online algorithm to perform real-time container placement. Online placement means executing the placement calculation when real-time requests arrive, and we can no longer perform a general partition on the DAG like *MDP*. Inspired by [6], we propose the **Greedy Step Placement (GSP)** algorithm to address *DSFP* (Algorithm 3). The mechanisms and procedure of GSP is given below.

*Mechanism One, GSP satisfies workflow's SLO requirements via satisfying each steps sub-SLO:* Because the arrival of dynamic functions is known only when the workflow reaches a certain step, GSP performs container placement on a step level. Inspired by [6], we reduce the constraint of satisfying a DAG's SLO (9) to satisfying each step's sub-SLO, denoted as

$$\sum_{s_m \in G} T_{s_m} \leq T_{SLO}. \quad (15)$$

Then, the problem becomes how to set each step's sub-SLO. Our insight here is that *when reaching the cut-off point of the (3) curve, the sharing-time ratio is the highest*. Using knowledge of  $G_{known}$ , for each step of it, GSP finds the function with the longest execution time at the variation point. Then, it greedily uses it to calculate the weight of a step, denoted by  $\theta$

$$\theta(s_m) = \beta t_{j,k}^T. \quad (16)$$

Here,  $\beta$  is a parameter for adjustment. A step's weight determines its portion of the SLO time. So the sub-SLO of a step  $s_m$  is calculated by

$$T_{s_m} = \frac{\theta(s_m)}{\sum_{f_j} \theta(s_m)} \cdot T_{SLO}. \quad (17)$$

Of course, this way of obtaining  $T_{s_m}$  overlooks the possible impact of functions within  $G_{dynamic}$ . This is a compromise made to address dynamic scenarios.

*Mechanism Two, GSP uses greedy strategy to co-place functions of a step:* With the sub-SLO of each step determined, we need to satisfy it while trying to maximize the sharing within each step. We still leverage Insight 1, trying to place function containers in one work-node. Requests of a step would arrive in two different ways. One is *in Batch* (e.g., several functions would be invoked with a minor delay after a fan-out node). Another is *With Delay*, where function requests of the same step do not arrive at the same time. To tackle this situation, GSP uses a greedy strategy. For batched requests, GSP simulates trying to place function containers in one work-node (Line 5-8). It calculates whether it exceeds the sub-SLO. If the maximum time exceeds the sub-SLO, *AddAndBalance()* selects a new work-node and evenly places containers in old and new work-nodes. Placement is performed for all functions afterward. For requests arriving with delays, GSP simply calculates if a placement would lead to an SLO violation. If it does, it selects a new work-node and places future containers in it. Placement is done as long as the request comes.

*Mechanism Three, GSP tends to utilize work-nodes used by preceding steps.* This aligns with Insight (2) mentioned earlier. Reusing work-nodes where preceding containers were placed increases the benefits of cache memory sharing.

The progression of GSP is rather straightforward. Given a partially known DAG and work-nodes, GSP first calculates the sub-SLO for each step (Lines 1-2). Next, it awaits the arrival of each step and selects work-nodes to place function containers. The placement follows Mechanisms Two and Three (Lines 4-13). The placement procedure concludes when requests of the workflow no longer arrive.

GSP leverages the methodology of the greedy algorithm on two levels: calculating sub-SLO for steps on the DAG level and making best efforts to place containers on the step level. The time complexity of GSP is  $O(|V| + |W|)$  as it only needs to parse the vertices and work-nodes several times. The space complexity is  $O(|V| + |W|)$ , accounting for storing the vertices and work-nodes. Note that GSP may not find an exact and optimal solution for problem *DSFP*, but it is locally optimal for such online scenarios.

TABLE I  
COMPARISON OF THREE ALGORITHMS

Algorithm	DAG Structure	Methodology
Optimal	Static	Exhaustion
MDP	Static	Score Based Sub-SLO Partition
GSP	Dynamic	Greedy Algorithm

### C. Solution Summary

A comparison of the three algorithms is presented in Table I. The Optimal Algorithm is an impractical one with a non-polynomial execution time, but can be used as a baseline. MDP and GSP individually handles static and dynamic DAGs. It is important to note that all three algorithms require prior knowledge of function execution time, implying that newly arrived function workflows may not be suitable for their application. To enhance accuracy and feasibility, we recommend employing MDP and GSP on workflows with a minimum of 20 invocation records. This ensures a more robust and reliable application of these algorithms in optimizing container placement and resource utilization. As for invocation recording, current open-source or commercial platforms all provide such abilities or plugins [3], [13].

## VI. IMPLEMENTATION

We have implemented MDP and GSP in the serverless platform OpenWhisk [13], which is the open-source project that serves not only commercial usage in IBM Cloud Functions but also many scientific researches. The source code of the system is in Github.<sup>1</sup>

The implementation of the Merged DAG Partition (MDP) and Greedy Step Placement (GSP) algorithms involves intricate modifications to OpenWhisk's core components, primarily focusing on the Controller, with additional adaptations to the Invoker and DB modules.

1) *Modified Invocation Recorder*: In the first phase of implementation, our focus is on augmenting the functionality of OpenWhisk Controller through a sophisticated invocation recorder. This enhanced recorder goes beyond merely logging each function's execution time; it also captures the system's resource pressure dynamics during execution. The recorded data serves a dual purpose: not only does it facilitate the generation of  $\xi(o_k, f_j)$ , crucial for determining variation points for each function, but it also contributes valuable insights into system behavior. In cases of clusters with homogeneous work-nodes, the recorder generates  $\xi(o_k, f_j)$  once for each function. As the diversity of work-node types increases, more curves are generated. However, given the tendency of cloud vendors to deploy similar virtual machine clusters, the record scale remains manageable. These records are stored in a dedicated CouchDB table within OpenWhisk, specifically designed for tracking workflow invocations. It's important to note that regular function invocations that do not form part of workflows continue to leverage the conventional activation recorder.

<sup>1</sup><https://github.com/Yorklandian/TPDS>

2) *Placement Algorithms (MDP and GSP)*: The second phase involves the actualization of MDP and GSP algorithms within OpenWhisk Controller, implemented using Scala. These algorithms are designed to collect real-time data and perform intricate placement calculations. Activation of MDP and GSP occurs in response to the arrival of workflow invocations. Input data crucial for these algorithms is sourced from the Controller Monitor, providing a comprehensive snapshot of each Invoker's status. The computed results are then communicated to the Load Balancer for precise dispatch operations. The status updates of Invokers are fed back to the Controller, initiating iterative processes and ensuring ongoing optimization.

Beyond the primary implementation, several auxiliary modifications have been incorporated into the platform. Three values are left configurable for users: **M** and **N** to assist in locating cut-off points for unusual situations, as stated in Section III, and  $G_{sub}^{lower}$  to control the tolerance of MDP and GSP. Specific values for these parameters will be provided in the evaluation setup sections below. An ImageComparer has been introduced, empowering the Controller to conduct layer-wise comparisons of function container images. This automated process streamlines the computation of (5). Additionally, the orchestration of Directed Acyclic Graphs (DAGs) is facilitated through OpenWhisk Composer and Conductor [13]. The static structure of a DAG is defined within Composer, while OpenWhisk Conductor introduces dynamic elements into workflows, adding flexibility to the overall system architecture. These enhancements collectively contribute to the robustness and efficiency of OpenWhisk in managing complex serverless workflows.

## VII. EVALUATION

We design several evaluations to answer the following research questions:

- How do MDP and GSP perform, comparing with original and optimal placement strategies? Can they effectively handle the trade-off between memory sharing and execution time deterioration? (Section VII-B)
- Are the SLO guaranteed while placing functions with MDP and GSP? How do they perform under different system pressures? (Section VII-C)
- How much memory could be saved due to layer-wise sharing-based function container placement, compared with non-sharing scenarios? (Section VII-D)
- What is MDP and GSP's sensitivity to the ratio of common layers and workflow's CPU pressure? (Section VII-E)

### A. Experiment Setup

1) *Workflow Selection*: To evaluate the performance of the algorithms, we selected and modified several serverless workflows from existing benchmarks [22], [27], [28]. These workflows represent various scenarios, including CPU-sensitive image processing, memory-sensitive tasks requiring large packages, and small workflows with balanced resource needs. They are real-world serverless benchmarks aimed for purposes including scientific computing and backend services [22], [28]. There are also other available benchmarks like DayDream [29], the reason



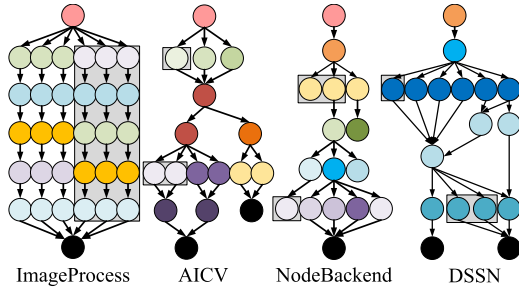


Fig. 10. Structure of four workflows.

we choose above benchmarks is that they focus on the serverless field and are adaptable to OpenWhisk.

We selected four representative workflows with the structure shown in Fig. 10, each serving real computational needs as real-life serverless applications. As benchmarks are originally static, they are designed with dynamic DAG alternations to test GSP. The dynamic parts of each workflow are contained in grey boxes in Fig. 10. *DYNAMIC* means these parts are NOT prior known to the serverless platform. OpenWhisk knows their existence in the workflow only when they are invoked during the progress of the workflow. The dynamic parts are created in two patterns. One is that the count of parallel sequences is not fixed, as ImageProcess. The other is that whether several branch functions would be invoked is unknown, like AICV and NodeBackend. Note that in later experiments, these dynamic functions would all DO exist and BE invoked. So that dynamic and static DAG would have the same function number and graph structure, not affecting the fairness for comparison between MDP and GSP.

- *ImageProcess*: This scientific workflow utilizes Python 3.6 with the Python library PIL for simple image processing in a parallel pattern to leverage FaaS platform's scalable features. Its dynamic version has a random number of parallel sequences ranging from 3 to 6.
- *AICV*: This workflow incorporates model-enhanced OpenCV features to handle AI-related computer vision requests, using Python 3.7 as the basic runtime. Its dynamic parts contains several branch functions whose absence would not break the main process.
- *NodeBackend*: Comprising several Node.js functions, this workflow forms a scalable serverless backend application which handles http requests. A dynamic version is made by having three branch functions' invocation decided by a random seed.
- *DeathStarSocialNetwork(DSSN)*: This workflow modifies several DeathStarBench [28] micro-services as serverless functions. Forming a simplified backend workflow. Its dynamic version follows the same principle of NodeBackend.

These selected workloads are first invoked 100 times under different system pressures and execution time curves are recorded and generated. Besides these selected workloads, dozens more workload are all tested when evaluating MDP and GSP's sensitivity. They'd be used for sensitivity analysis on MDP and GSP's performance reacting to the ratio of workflow functions' common layers and workflow's CPU pressure. Based

on the recorded values, we set  $M$  as 80% and  $N$  as 3 to alleviate high pressure or execution time. Among all the workflows, averagely 31.2% of image layer size could be shared. Therefore, we set a rather tolerant  $G_{sub}^{lower}$  as 25% to better observe the performance of MDP and GSP. It is important to note that a higher  $G_{sub}^{lower}$  would make MDP and GSP more stringent on memory sharing, resulting in lower sharing but better SLO guarantee.

2) *Baselines*: To test the effectiveness of MDP and GSP, we selected five container placement strategies as baselines:

- *Wisefuse* [11] places concurrent function containers in a fair manner by evenly distributing containers across all work-nodes, minimizing the pressure on each machine but disregarding memory sharing.
- *Faastlane* [9] places all function containers of a workflow into a single work-node, optimizing memory footprint at the expense of work-node pressure considerations.
- *Phontons* [30] co-locates parallel invocations together to improve memory utilization, not to meet latency or cost objectives.
- *RainbowCake* [31] leverages layer-wise memory sharing for reducing cold starts and memory consumption. We combine its algorithm and OpenWhisk's default scheduling as a baseline.
- *NP-Optimal (NPO)* performs optimal placement using Algorithm 1. While not practical in real life due to its NP-Hard nature, we calculated the results in advance for comparison with MDP and GSP.

3) *Work-Node Setup*: Our experiments were conducted on an OpenWhisk Cluster deployed using Kubernetes. The cluster contains 10 Ubuntu 20.04 servers, each equipped with 64 GB of memory and 20 CPUs. We utilized the Linux command tool *STRESS* [32] to regulate the occupancy of work-nodes' CPU and memory resources, creating a FaaS cluster environment that closely mimicked real production FaaS platforms [10].

*STRESS* directly adjusts the work-node's CPU occupancy by generating background tasks, enabling us to control the system pressure of work-nodes. However, the system pressure set by *STRESS* is relatively static. A real serverless work-node would, in fact, exhibit dynamic system pressure levels, with functions spawning and terminating. In this scenario, in addition to setting a basic system pressure with *STRESS*, we employ scripts to create CPU-consuming functions running in the background, simulating a more realistic serverless environment. For instance, when we specify a work-node's system pressure as 30%, we actually use *STRESS* to establish a static system pressure of 25%. Concurrently, several CPU-consuming functions would be randomly invoked, consuming 5% to 10% of CPU resources. This approach of setting system pressure is both more realistic and convenient.

## B. Regular Evaluation for MDP and GSP

We first tested the two algorithms under a simulated in-production serverless cluster environment. Inspired by [10], we randomly set work-nodes to have resource occupancy between 40% to 60% as the initial state of the cluster, creating a relatively

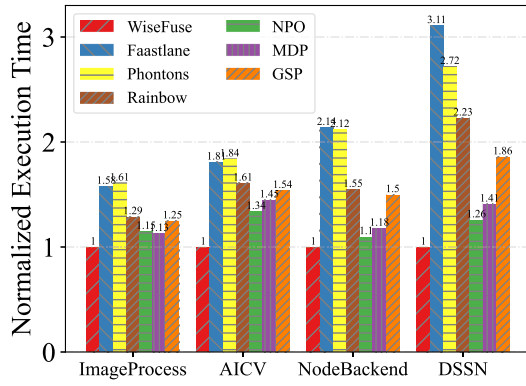


Fig. 11. Workflow execution time comparison.

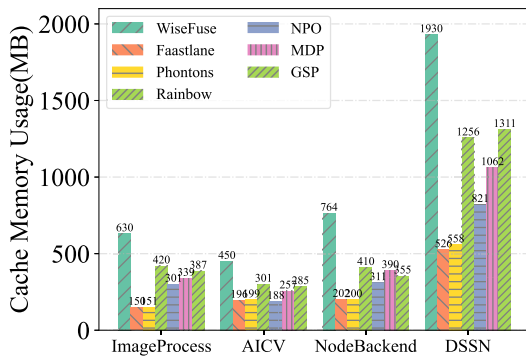


Fig. 12. Cache memory usage comparison.

busy serverless cluster. OpenWhisk first performed container placement using the baseline policies and their results were recorded. We then set the SLO for MDP and GSP using the mean execution time of Faastlane and SA. Next, the four workflows were invoked dozens of times in the cluster using Optimal, MDP, and GSP. For GSP, we made several parts of the workflows dynamic as stated above.

We gathered data on the average execution time and page cache memory consumption of the workflows. As functions generally exhibit similar runtime memory consumption, we focused on presenting buffer/cache memory metrics. The comparisons are visualized in Figs. 11 and 12. SA, Faastlane, and Phontons represented the extremes of the memory-time spectrum. SA displayed the lowest execution time but the highest cache memory occupancy, while Faastlane and Phontons exhibited the opposite. SA places functions in as many different machines as possible while Faastlane and Phontons do quite the opposite. For workflows with a large number of functions like Death-StarSocialNetwork, their consumption of buffer/cache memory and execution time would distinguishing differ. MDP and GSP achieved a balanced state, with MDP slightly outperforming GSP. MDP's placement conserved more memory than the greedy GSP, which performed real-time placement. Rainbow-Cake could also reach a balanced state alias to MDP and GSP. But as its scheduling lacks DGA partition based mechanism, its memory consumption and overall execution time are both

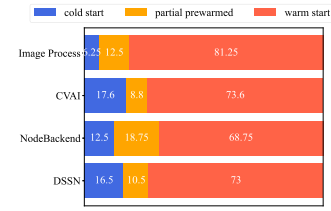


Fig. 13. Warm/Cold start rate for MDP.

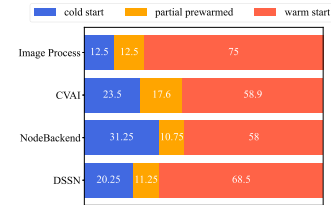


Fig. 14. Warm/Cold start rate for GSP.

higher than MDP and GSP. Statically, MDP averagely saves up to 63% of the cache/buffer memory occupancy comparing with SA. Furthermore, GSP's execution time closely aligned with the SLO, whereas MDP's execution time increased less than GSP, demonstrating MDP's conservative approach in not overburdening work-nodes. Moreover, MDP's results closely approximated the Optimal, indicating satisfactory outcomes within polynomial time.

In addition, we collected serverless-specific metrics for MDP and GSP, focusing on the cold start rate. Since both MDP and GSP possess the capability to preprocess the placement of function containers upon the first workflow invocation, subsequent function containers can be pre-warmed in the work-nodes they are destined to occupy. This pre-warm strategy was employed in our experiments. Without adopting any additional pre-warm strategies and utilizing OpenWhisk's original 10-minute keep-alive time, both MDP and GSP effectively reduced the workflow's cold start rate.

Figs. 13 and 14 illustrate the cold start rate of the four workflows when newly invoked in the clusters. Partially prewarmed containers refer to containers whose invocation occurs before the prewarming process concludes. In general, both MDP and GSP demonstrated the capability to decrease the cold start rate of workflows. GSP's cold start rate is higher due to the inability to prewarm functions in the dynamic parts. Since reducing cold start is not the main focus of this paper, no further comparisons or discussions on this aspect will be provided.

### C. Evaluation for SLO Guaranteeing

The pivotal concern for MDP and GSP lies in ensuring the Service Level Objectives (SLO) of workflows. To assess this, we manipulated the cluster's average system pressure, ranging from 30% to 70%, simulating various busy statuses within the serverless cluster. Note that they are all in fact dynamic pressures as stated in Section VII-D. Additionally, we expanded the SLO constraint to 1.5 times the original. Subsequently, we executed

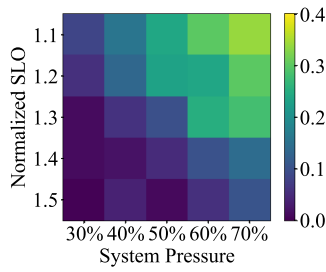


Fig. 15. ImageProcess SLO violation (MDP).

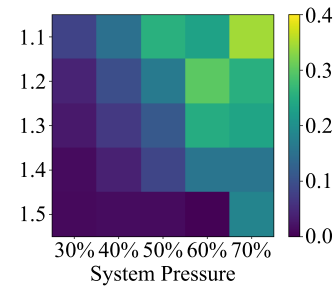


Fig. 19. ImageProcess SLO violation (GSP).

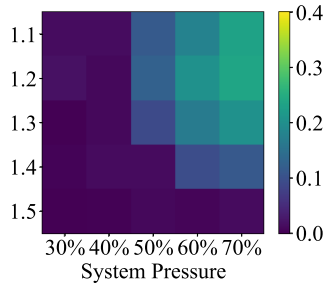


Fig. 16. AICV SLO violation (MDP).

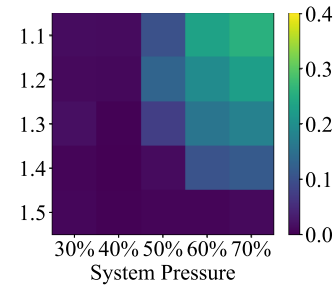


Fig. 20. AICV SLO violation (GSP).

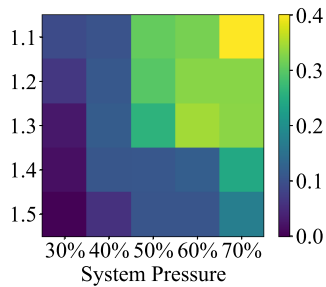


Fig. 17. NodeBackend SLO violation (MDP).

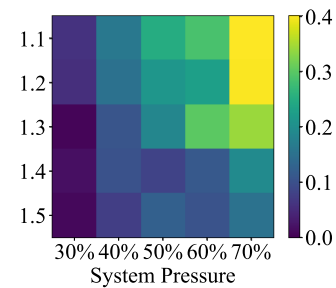


Fig. 21. NodeBackend SLO violation (GSP).

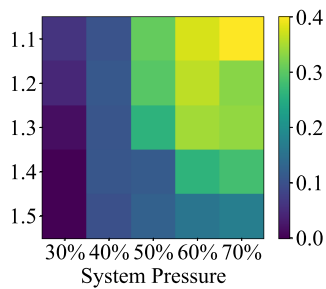


Fig. 18. SocialNetwork SLO violation (MDP).

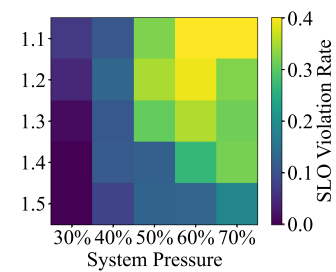


Fig. 22. SocialNetwork SLO violation (GSP).

multiple instances of the four workflows, with MDP and GSP orchestrating container placement within the cluster. Violation rates were meticulously recorded under each condition, revealing insights presented from Figs. 15 to 18 for MDP and Figs. 19 to 22 for GSP.

As depicted in the graphs, both MDP and GSP performed admirably, showcasing acceptable SLO violation rates ranging from 0 to 0.4, contingent upon system pressure and SLO time

requirements. Variances in SLO violation distributions were observed among different workflows in the heat maps. ImageProcess exhibited a relatively even distribution, where an increase in system pressure and a reduction in SLO time both contributed to a noticeable rise in violation rates. AICV displayed heightened sensitivity to SLO time after the system pressure reached 50%, while NodeBackend demonstrated greater sensitivity to system pressure after the SLO time dropped below 1.3. Notably,



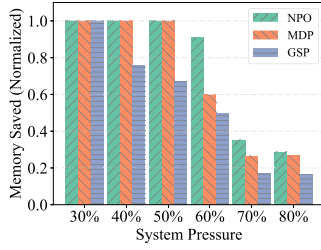


Fig. 23. Memory saving for ImageProcess.

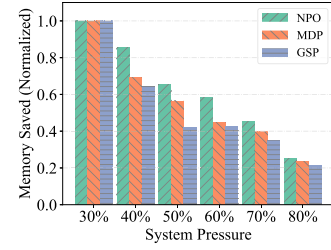


Fig. 26. Memory saving for SocialNetwork.

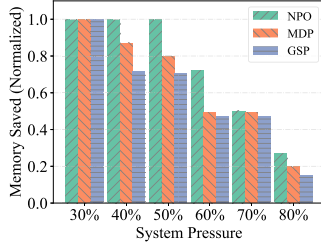


Fig. 24. Memory saving For AICoComputerVision.

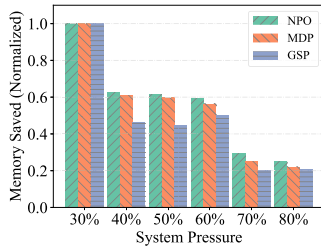


Fig. 25. Memory saving for NodeBackend.

MDP and GSP excelled within their respective scenarios, with discernible differences in SLO violation rates, showcasing instances where GSP's results lagged behind MDP. The reason GSP performs slightly worse than MDP is attributed to its inability to account for the impact of dynamic functions, thus rendering its placement less optimal. However, it's important to note that the deterioration is not significant, as GSP's greedy strategy still functions well locally.

#### D. Evaluation for Cache Memory Saving

MDP and GSP aimed to optimize cache memory usage, thereby reducing the overall memory consumption of the cluster. Experiments were conducted to observe the cache/buffer memory-saving effects, as depicted in Figs. 23, 24, 25 and 26. In these figures, the original cache memory is the total cache/buffer memory consumption of the workflow when scheduled using WiseFuse (630 MB, 450 MB, 764 MB and 1930 MB). We selected this metric because WiseFuse evenly places containers, leading to the highest consumption of cache memory. The "Memory Saved" metric is the original cache memory size reduces cache memory size achieved by our placement strategies. These values are normalized for better comparison. The illustrations of figures revealed the statics of buffer/cache memory

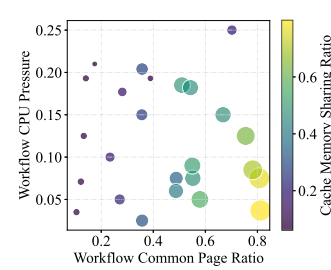


Fig. 27. Sensitivity of MDP.

conservation in response to increasing system pressure. Notably, as system pressure escalated, both MDP and GSP exhibited a tendency to distribute containers across more work-nodes, mitigating SLO violations but concurrently leading to a reduction in saved cache/buffer memory. Despite the saved memory not reaching the Optimal value, both MDP and GSP demonstrated efficacy.

MDP achieved remarkable memory savings, reaching up to 89.5% compared to the Optimal strategy. Even under high pressure, GSP achieved a commendable 63% memory savings compared to the Optimal strategy. It's important to note that the introduction of dynamic contexts introduced an element of uncertainty, leading to suboptimal results. Integrating these findings with the analysis in (Section VII-C), it can be inferred that both MDP and GSP prioritized SLO adherence while striving to maximize memory sharing.

#### E. Sensitivity Analysis

In this section, we assess the sensitivity of MDP and GSP. The impact of MDP and GSP, particularly in terms of saving buffer/cache memory, relies on the sharing of container image layers among workflow functions. However, the summing of CPU pressure by placing functions together may prevent MDP and GSP from doing so. To examine how this influences the performance of MDP and GSP, we conducted additional experiments. The work-nodes were configured to have resource occupancy between 40% and 60%, as described in Section VII-B. In addition to the selected workflows mentioned earlier, we included another 21 workflows obtained from [22], [27], [28]. In total, we had 25 workflows, covering a range of common page ratios and CPU pressures displayed in Fig. 27. The common page ratio represents how much pages of a workflow

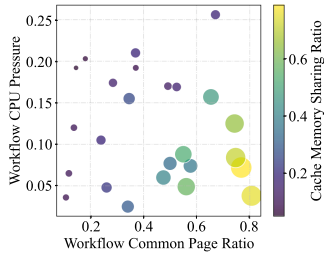


Fig. 28. Sensitivity of GSP.

are the same, using page size for calculation. The workflow CPU pressure is the workflow's total pressure to the cluster.

These workflows were processed and invoked, and we collected data of their Cache Memory Sharing Ratio, representing the percentage of buffer/cache memory shared, which would mean memory saving. The results are presented in Fig. 27 for MDP and Fig. 28 for GSP. The color and size of the scatter plots represent the values of the Cache Memory Sharing Ratio. Fig. 28 differs slightly from MDP as dynamic functions are not pre-calculated. Clearly, workflows with low overall CPU pressure and a high common page ratio benefit the most from MDP and GSP. We recommend that workflows with such characteristics actively leverage MDP and GSP. The common page ratio should better be more than 50%. Moreover, if the average CPU pressure in the serverless cluster is not high, workflows with low CPU pressure and a high common page ratio would also derive significant benefits from these strategies.

#### F. Evaluation Summary

In summary, MDP and GSP consistently achieved a balanced state, optimizing both memory sharing and SLO adherence in placing function containers within a cluster. The inter-container memory sharing within a Directed Acyclic Graph (DAG) emerged as a crucial factor. Additionally, both algorithms demonstrated resilience against the impact of system pressure, effectively navigating potential deterioration in performance.

### VIII. RELATED WORK

*Locality of serverless computing:* The optimization of serverless platforms through the strategic utilization of data and function locality has emerged as a significant area of research [5]. Abad et al. introduced a package-aware scheduling algorithm aimed at assigning functions requiring the same package to the same worker node [33], a concept further explored by Aumala et al. in [34]. Alexander and Prateek delved into a trade-off analysis involving locality, load distribution, and randomness in function placement [35]. Palette, introduced by Abdi et al. empowers users to manually control the locality of FaaS Apps, demonstrating improved performance by eliminating cold starts [36]. However, these studies often overlook the potential benefits of memory sharing between functions, and the impact on CPU pressure is frequently disregarded.

*Memory sharing and deduplication:* Prior research has concentrated on memory deduplication within FaaS platforms.

Bravo et al. conducted a survey on library sharing among containers [37]. Qiu explored the potential and mechanisms of memory sharing for serverless functions [38]. Li et al. designed TETRIS, a framework leveraging memory sharing for inference services, resulting in significantly reduced memory usage [39]. Saxena et al. proposed a novel approach to identify memory redundancy and reduce duplicated memory footprint [36]. FaaSPIPE proposes a serverless workflow runtime leveraging a simplified distributed shared memory, achieving high reduction in workflow latency less network traffic [40]. Andrea et al. introduces a container-based serverless architecture with a shared-memory approach and message-oriented middleware [41]. However, these initiatives often overlook the computational pressure introduced by sharing and fail to address the crucial trade-off between sharing and performance. Additionally, the profiling of runtime memory blocks, a common practice in the sharing approaches above, may pose safety concerns and result in extended processing times.

*Scheduling, DAG partitioning, and SLO-memory trade-off:* Carver et al. introduced Wukong, utilizing decentralized scheduling for parallel task scheduling [42]. Wisefuse [11] bundles and fuses vertices in Serverless Directed Acyclic Graphs (DAGs) to minimize data transfer latency and enhance performance. Faastlane [9] accelerates FaaS DAG execution by invoking functions within a single Virtual Machine (VM). Aquatope [43] employs Bayesian Optimization (BO) to determine the optimal resource configuration. Photons [30], StepConf [6], Li [44] and Astra [45] aim to balance function memory configuration and overall Service Level Objective (SLO) using heuristic, AI-based, or prediction methods.

In comparison, our contribution actively harnesses sharing among serverless functions while proactively addressing performance degradation resulting from work-node pressure. We provide innovative solutions to manage this trade-off, achieving a resource consumption footprint that satisfies the required Service Level Objectives (SLO). Through our approach, we navigate the delicate balance between efficient resource utilization and guaranteed performance standards.

### IX. CONCLUSION

In this paper, we propose a system built upon OpenWhisk that implements MDP and GSP, two container placement algorithms specifically tailored for both static and dynamic serverless workflows. These algorithms take into account considerations of memory sharing and SLO requirements. MDP, boasting an approximation ratio of 2, leverages the sharing locality inherent in serverless workflows by strategically partitioning the Directed Acyclic Graph (DAG) into sub-graphs. On the other hand, GSP adopts a greedy strategy at both the graph and step levels to co-locate containers effectively. Our evaluations on the OpenWhisk platform involved multiple well-constructed workflows. Comparing the results with baselines, we observe that both algorithms achieve a harmonious balance between memory sharing and SLO guaranteeing. Noteworthy memory savings are realized while ensuring that the workflow's SLO

is consistently met. Additionally, the design of both algorithms ensures polynomial time complexity, making them efficient and practical for real-world implementations.

## REFERENCES

- [1] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," 2019, *arXiv: 1902.03383*.
- [2] Amazon, "AWS lambda," 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/>
- [3] Microsoft, "azure functions," 2024. [Online]. Available: <https://azure.microsoft.com/en-us/products/functions/>
- [4] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.
- [5] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–34, 2022.
- [6] Z. Wen, Y. Wang, and F. Liu, "StepConf: SLO-aware dynamic resource configuration for serverless function workflows," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1868–1877.
- [7] Docker, "Docker," 2024. [Online]. Available: <https://www.docker.com/>
- [8] M. Szeredi, "overlay," 2024. [Online]. Available: <https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html>
- [9] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 805–820.
- [10] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "SpecFaaS: Accelerating serverless applications with speculative function execution," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2023, pp. 814–827.
- [11] A. Mahgoub et al., "WISEFUSE: Workload characterization and DAG transformation for serverless workflows," in *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, pp. 1–28, 2022.
- [12] S. Zhang, Y. Guo, Z. Guo, H. Hu, and G. Chen, "SMAF: A secure and makespan-aware framework for executing serverless workflows," *Sci. China Inf. Sci.*, vol. 66, no. 3, 2023, Art. no. 139105.
- [13] Apache, "apache openwhisk," 2022. [Online]. Available: <https://openwhisk.apache.org/>
- [14] H. Kellerer, U. Pferschy, D. Pisinger, H. Kellerer, U. Pferschy, and D. Pisinger, *Multidimensional Knapsack Problems*. Berlin, Germany: Springer, 2004.
- [15] Google, "Google cloud functions," 2024. [Online]. Available: <https://cloud.google.com/functions>
- [16] Linux, "Docker overlayfs," 2024. [Online]. Available: <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>
- [17] B. Hou, S. Yang, F. A. Kuipers, L. Jiao, and X. Fu, "EAVS: Edge-assisted adaptive video streaming with fine-grained serverless pipelines," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2023, pp. 1–10.
- [18] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1288–1296.
- [19] J. Jiang et al., "Towards demystifying serverless machine learning training," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 857–871.
- [20] Azure, "azure durable function," 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-functions/durable/>
- [21] Amazon, "AWS step functions," 2024. [Online]. Available: <https://aws.amazon.com/step-functions/>
- [22] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *Proc. IEEE 12th Int. Conf. Cloud Comput.*, 2019, pp. 502–504.
- [23] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 303–320.
- [24] OpenWhisk, "OpenWhisk CPU allocation," 2023. [Online]. Available: <https://github.com/apache/openwhisk/pull/5443>
- [25] C. Chekuri and S. Khanna, "A polynomial time approximation scheme for the multiple knapsack problem," *SIAM J. Comput.*, vol. 35, no. 3, pp. 713–728, 2005.
- [26] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 153–167.
- [27] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 30–44.
- [28] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 3–18.
- [29] R. B. Roy, T. Patel, and D. Tiwari, "DayDream: Executing dynamic scientific workflows on serverless platforms with hot starts," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 1–18.
- [30] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a diet," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 45–59.
- [31] F. Yu, L. Wang, and X. Park, "RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," 2024. [Online]. Available: <https://intellisys.haow.ca/assets/pdf/hanfeiasplos24spring.pdf>
- [32] Linux, "linux stress tool," 2023. [Online]. Available: <https://pkgs.org/download/stress>
- [33] C. L. Abad, E. F. Boza, and E. Van Eyk, "Package-aware scheduling of FaaS functions," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 101–106.
- [34] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," in *Proc. 19th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2019, pp. 282–291.
- [35] A. Fuerst and P. Sharma, "Locality-aware load-balancing for serverless clusters," in *Proc. 31st Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2022, pp. 227–239.
- [36] M. Abdi et al., "Palette load balancing: Locality hints for serverless functions," in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 365–380.
- [37] J. Bravo Ferreira, M. Cello, and J. O. Iglesias, "More sharing, more benefits? A study of library sharing in container-based infrastructures," in *Proc. 23rd Int. Conf. Parallel Distrib. Comput.*, Santiago de Compostela, Spain, Springer, 2017, pp. 358–371.
- [38] W. Qiu, "Memory deduplication on serverless systems," Master's thesis, Dept. Inf. Technol. Elect. Eng., ETH Zurich, Zürich, Switzerland, 2021.
- [39] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 473–488.
- [40] R. Tong, "FaaSPIPE: Fast serverless workflows on distributed shared memory," in *Proc. IFIP Int. Conf. Netw. Parallel Comput.*, Springer, 2022, pp. 83–95.
- [41] A. Sabbioni, L. Rosa, A. Bujari, L. Foschini, and A. Corradi, "A shared memory approach for function chaining in serverless platforms," in *Proc. IEEE Symp. Comput. Commun.*, 2021, pp. 1–6.
- [42] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 1–15.
- [43] Z. Zhou, Y. Zhang, and C. Delimitrou, "AQUATOPE: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 1–14.
- [44] Z. Li, H. Yu, and G. Fan, "Time-cost efficient memory configuration for serverless workflow applications," *Concurrency Comput.: Pract. Exp.*, vol. 34, no. 27, 2022, Art. no. e7308.
- [45] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Astra: Autonomous serverless analytics with cost-efficiency and QoS-awareness," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 756–765.



**Dazhao Cheng** (Senior Member, IEEE) received the BS and MS degrees in electrical engineering from the Hefei University of Technology, in 2006, and the University of Science and Technology of China, in 2009, and the PhD degree from the University of Colorado at Colorado Springs, in 2016. He was an AP with the University of North Carolina at Charlotte, in 2016–2020. He is currently a professor with the School of Computer Science, Wuhan University. His research interests include Big Data and cloud computing.





**Kai Yan** received the BS degree in software engineering from Wuhan University, in 2023. He is currently working toward the MS degree in software engineering with Wuhan University. His research interests include serverless computing, distributed systems and virtualization.



**Yili Gong** received the BS degree in computer science from Wuhan University, in 1998, and the PhD degree in computer architecture from the Institute of Computing, Chinese Academy of Sciences, in 2006. She is currently an associate professor with the School of Computer Science, Wuhan University. Her research interests include intelligent operations and maintenance in HPC environments, distributed file systems, and blockchain systems.



**Xinquan Cai** received the BS degree in computer science and technology from ChongQing University, in 2021. He is currently working toward the PhD degree in computer science with Wuhan University. His research interests include serverless computing and virtualization.



**Chuang Hu** (Member, IEEE) received the BS and MS degrees from Wuhan University, in 2013 and 2016, respectively, and the PhD degree from the Hong Kong Polytechnic University, in 2019. He is currently an associate researcher with the School of Computer Science, Wuhan University. His research interests include edge learning, federated learning/analytics, and distributed computing.