ESG: Pipeline-Conscious Efficient Scheduling of DNN Workflows on Serverless Platforms with Shareable GPUs

Xinning Hui
xhui@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Yuanchao Xu
yxu314@ucsc.edu
University of California, Santa Cruz
Santa Cruz, USA

Zhishan Guo
zguo32@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Xipeng Shen
xshen5@ncsu.edu
North Carolina State University
Raleigh, NC, USA

ABSTRACT

Recent years have witnessed increasing interest in machine learning inferences on serverless computing for its auto-scaling and cost effective properties. Existing serverless computing, however, lacks effective job scheduling methods to handle the schedule space dramatically expanded by GPU sharing, task batching, and inter-task relations. Prior solutions have dodged the issue by neglecting some important factors, leaving some large performance potential locked. This paper presents ESG, a new scheduling algorithm that directly addresses the difficulties. ESG treats sharable GPU as a first-order factor in scheduling. It employs an *optimality-guided adaptive* method by combining A*-search and a novel *dual-blade pruning* to dramatically prune the scheduling space without compromising the quality. It further introduces a novel method, *dominator-based SLO distribution*, to ensure the scalability of the scheduler. The results show that ESG can significantly improve the SLO hit rates (61%-80%) while saving 47%-187% costs over prior work.

CCS CONCEPTS

• **Computer systems organization** → **Cloud Computing**; • **Computing methodologies** → **Planning and scheduling**.

KEYWORDS

Cloud computing, Serverless Computing, Quality of Service, Function-as-a-Service, Resource Management, Resource Allocation, Resource Efficiency, Machine Learning for Systems, Deep Learning

ACM Reference Format:
Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. 2024. ESG: Pipeline-Conscious Efficient Scheduling of DNN Workflows on Serverless Platforms with Shareable GPUs. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3625549.3658657

HPDC '24, June 3–7, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06
https://doi.org/10.1145/3625549.3658657

1 INTRODUCTION

Recent years have witnessed a rapidly growing interest in machine learning (ML) inferences on serverless platforms, thanks to the ease of programming and maintenance, autoscaling, and pay-as-you-go billing of serverless computing [17, 19, 27, 57–59].

Due to the computing-intensive nature of ML inferences, it is desirable to provide GPU support on serverless platforms to ML workloads, which can help significantly boost the service throughput for the massive parallelism of GPUs [23, 24]. However, the current state of commercial serverless platforms (e.g., AWS Lambda [2], Google Cloud Function [7], Azure Function [3], and Knative [8]) is falling behind: The schedulers and resource management are still CPU-centric, oblivious to GPU resources.

There have been some recent research efforts toward adding GPU support to ML on serverless platforms. Some approaches have leveraged NVIDIA MPS [10] to facilitate GPU sharing across distinct function instances [22, 30, 58]. Others have introduced techniques to enable batching for ML inference, thus increasing overall throughput [16, 58]. Additionally, certain studies have extended scheduling algorithms to manage heterogeneous computing resources more effectively [25, 56].

Despite the progress made so far, two challenges remain, which have kept much of the potential of GPU out of reach for serverless ML.

(i) **The dramatically expanded search space for scheduling.** An important step in serverless scheduling is to *configure the serverless functions*, that is to determine the amount of resource to assign to each of the serverless functions in an application to meet the Service Level Objective (SLO) while consuming the minimum resource. The configuration space is as large as $n = m^k$, where m is the number of resource allocation options for one serverless function in the application, and k is the number of functions in the application. Without considering sharable GPUs, the options for a function are just the number of vCPUs to consider. With sharable GPUs, the space becomes three-dimensional: *(batch size, number of vCPUs, number of vGPUs)*. The "batch size" here refers to the number of invocations of a serverless function to be grouped into one batch to run. Batching is essential for GPU throughput due to its massive parallelism. The search space hence expands cubically, from $n = m^k$ to $n = (m^k)^3$ (assuming the same number of options in all dimensions and a function can have its own options values). For $m = 5, k = 7$, the space increases from 78K options to 476 trillions.

Table 1: Comparison of serverless systems

Features	INFless	Fast-GShare	Orion	Aquatope	ESG
GPU sharing	√	√	×	×	√
Inter-function relation	×	×	√	√	√
Adaptive sched.	√	√	×	×	√
Data locality	×	×	×	×	√
Pre-warming	√	×	√	√	√

(ii) **The significantly increased complexity in handling performance variations.** Because the number of different states of available resources is multiplied by the available vGPUs, it calls for an adaptive scheduling algorithm to handle them properly. Performance variation is an inherent problem in serverless computing, where the running times of a serverless function vary much across invocations [43]. Batching exacerbates it as unpredictable job arrival times cause variations in the waiting time for forming a batch. Along with these variations, the system workload and resource availability on a serverless system show constant changes. Therefore, adapting agilely to dynamic changes and variations becomes crucial for a scheduler.

Previous studies have all dodged those challenges by neglecting some important dimensions of the problem (Table 1), causing a substantial loss in the quality of scheduling. These dimensions include **inter-function relations**, **GPU sharing**, and **runtime system state variations**. Specifically, based on the neglected dimension(s), previous solutions fall into one of two groups. (i) Those works *neglecting the relations between functions*, represented by Fast-GShare [30] and INFless [58], two most recent studies on sharable GPU support for serverless ML. An ML-based application often consists of multiple stages of work, which form a pipeline or a directed acyclic graph (DAG) [36]. A chatbot, for example, responds to a user's input by going through stages of speech recognition, natural language understanding, speech synthesis, and so on. The SLO of an AI-based application is usually for the *end-to-end* latency of the entire process. The schedule and configuration of one stage hence affect other stages, such as how much latency those stages can still have to allow the end-to-end time to meet the SLO, how much resource used in total, whether the data transfer is local or remote, and so on. Neglecting the inter-stage relations to reduce the scheduling complexity is hence not ideal, causing a large loss in quality (36%-61% as Section 5.1 shows). (ii) Those that *neglect GPU sharing and runtime variations*, represented by Orion [43] and Aquatope [61]. These proposals regard a GPU as a single device attached to a CPU and schedule jobs purely based on CPU and memory availability. Moreover, they are rigid: They pick a good configuration for every stage of the application before its execution, and then stick to the configuration throughout that execution, regardless of whether the actual latencies of some functions differ substantially from the expectation or how the resource availability changes as the execution reaches later stages. The strategy also leads to a large loss in scheduling quality (46%-80% as Section 5.1 shows).

Addressing the limitations is essential for unlocking the full potential of GPU for serverless ML. It is, however, not easy: The solution must be efficient enough to handle the enormous configuration space and agile enough to adapt to the dynamic changes of the system.

This paper presents our solution, a new scheduling algorithm named ESG (which stands for Efficient Serverless Scheduling for Sharable GPUs), which, for the first time, addresses all those challenges at the same time. ESG treats sharable GPU as a first-order factor in scheduling. For scheduling efficiency, ESG employs A*-search and introduces a *dual-blade pruning* to dramatically prune the huge search space of schedules without compromising the quality. For scalability, ESG further introduces *dominator-based SLO distribution* to prevent space explosion. For the quality of the scheduling results, ESG takes an *optimality-guided adaptive approach*. Rather than deciding the resource assignment of all functions in a workflow at the beginning and keeping it unchanged throughout the DAG execution as previous solutions [43, 44] do, ESG revisits and adjusts the schedules before the dispatch of every serverless function to adapt to the dynamic changes of the environment and the performance variability of the functions.

We have evaluated ESG on a series of workloads involving six DNNs and compared the performance with four state-of-the-art serverless scheduling solutions, INFless [58], FaST-GShare [30], Orion [43] and Aquatope [61]. The results show that the new scheduling algorithm ESG is effective in addressing the new challenges, consistently achieving the highest SLO hit rates with significantly lower resource usage, exhibiting a notable 61%-80% improvement in SLO hit rates and saving 47%-187% in costs than INFless and Fast-GShare, especially in challenging scenarios.

Overall this work makes the following major contributions:

- It provides ESG, the first scheduling algorithm that simultaneously tackles inter-function relations, GPU sharing, batching, and runtime variations.
- It introduces a set of novel optimizations to ensure high efficiency and scalability of the scheduling algorithm.
- It empirically confirms the effectiveness of ESG by comparing it with the state of the art represented by four previous scheduling algorithms.

2 BACKGROUND

Serverless architecture. We use OpenWhisk [11] as an example to explain the common architecture of serverless computing. OpenWhisk is an open-source, distributed Serverless platform that executes functions [1] in response to events at any scale. Figure 1 shows the architecture of OpenWhisk [1, 54]. It has a RESTful API, through which, users can create actions, invoke actions and check the action status. NGINX translates the command and forwards it to the Controller. Controller is where task scheduling happens. It maintains the health and remaining capacity of each Invoker (i.e., a computing node), tracks the warm instances on each Invoker, decides the resource assignments to tasks, and assigns tasks to Invokers. Then the Controller sends the invocation message to the selected Invoker via distributed messaging component (Kafka). The Invoker creates an execution environment (Docker container) after receiving the invocation message and manages its runtime (including stopping the container). OpenWhisk sets a fixed 10-minute

[1] "Function" in this paper refers to Serverless Function.

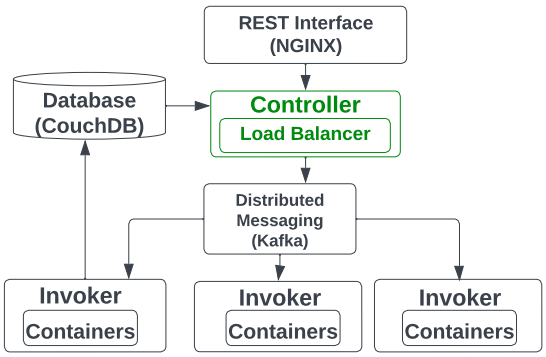


Figure 1: OpenWhisk architecture. Controller is where scheduling happens.

keep-alive policy for each instance; the Invoker informs the Controller when it unloads a container [54].

Scheduling of Serverless Functions. There are two main steps in scheduling serverless functions.

The first step is resource assignment, which determines the amount of resources assigned to a function. It is based on the memory requirement specified in the configuration, which is converted by some platforms into a number of vCPUs (each vCPU is associated with a fixed amount of memory). The scheduler ultimately determines the amount of resources to be assigned to that instance, which may be more than required to achieve better performance. OpenWisk uses the required amount by default, while recently schedulers [15, 26, 43, 48, 61] used a more sophisticated algorithm to decide the assignment of resources.

The second step is to map a function instance to an Invoker (e.g., a computing node). The controller maintains the available CPU resources of each Invoker according to its memory usage. In OpenWhisk, the controller picks one of the Invokers based on the hash value calculated by the namespace and action of the function. The generated index is called "home-invoker" [11], which is the invoker where the future instances of the function will reside by default. But if that invoker becomes unhealthy or lacks capacity, the scheduler will look for other invokers through a deterministic search. The mapping scheme is designed to get more warm starts.

GPU Sharing. GPU sharing is to allow multiple processes to share a single GPU for their executions. Modern GPUs support time sharing (one after the other) and spatial sharing. Spatial sharing is essential for ML inferences because an inference by an ML model often uses only a fraction of the massive parallel computing capacity of a GPU. Allowing multiple processes to execute concurrently on a GPU is essential for turning its computing power into throughput. Modern GPUs from NVIDIA offer two mechanisms for spatial sharing, Multi-Process Service (MPS) [10] and Multi-Instance GPU (MIG) [9]. MIG gives better isolations between partitions. MIG's ability to partition a single GPU into multiple hardware-isolated instances not only maximizes resource utilization but also significantly bolsters security, a critical aspect in today's cloud computing landscape. Although the size of a MIG partition is fixed at system booting time,

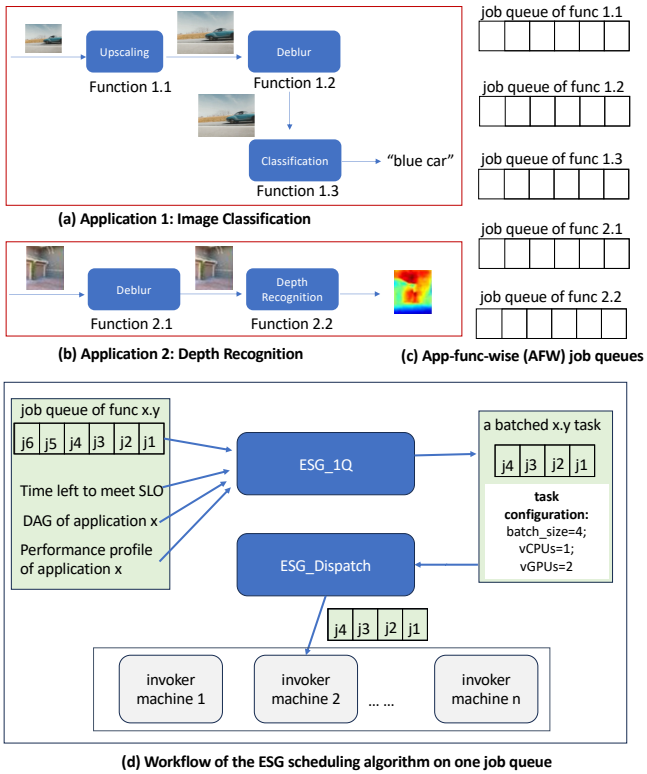


Figure 2: The app-func-wise (AFW) job queues of two example ML-based applications, and the ESG algorithm workflow in handling one job queue.

one application or serverless function may use more than one MIG partition by launching multiple GPU kernels concurrently.

3 SOLUTION: ESG SCHEDULING ALGORITHM

This section presents ESG, our GPU sharing-aware scheduling algorithm that is designed to respond to the unsolved challenges from inter-function relations and dynamic changes of resources.

3.1 Overview

ESG runs on the Controller of a serverless platform. Its objective is to maximize the Service Level Objective (SLO) hits of DNN inferences—that is, make as many DNN inferences deliver their results within the SLO latency as possible—while minimizing resource consumption (and hence cost). The problem is intuitive to understand, so a formal definition is omitted here for the sake of space. Interested readers may see the Appendix A for a formal rigorous definition of the problem.

For easy explanation, Figures 2 (a) and (b) illustrate two serverless ML-based applications in the form of DAG. Application 1 uses three DNN serverless functions in a sequence to first upscale an input image, then deblur it, and finally classify it. Application 2 uses two DNN serverless functions to first deblur an input image and then generate a depth image from it.

ESG introduces *application-function-wise (AFW) job queues* to group requests for the same serverless function of the same application together. The five serverless functions in Figure 2 (a) and (b) each have their own AFW job queue, as illustrated in Figure 2 (c). Notice that even though the same Deblur function is used in both applications, two AFW queues are created, one for each. This design allows ESG to put the functions of the same application on the same machine when possible to reduce the overhead of communications between functions (Section 3.4). The AFW queues reside on the Controller. Each of them gets populated as user requests arrive or its predecessor functions produce some triggering outputs.

Two-step Design. Figure 2 (d) outlines the high-level workflow of the two core algorithms of ESG. The Controller examines the job queues in a round-robin manner. If a job queue is ready to be scheduled, *ESG_1Q*, the first core algorithm of ESG, figures out an appropriate configuration for some jobs in that queue. The configuration includes (i) *batch size*: the number of jobs to be scheduled as a group—which we call a *task*; (ii) *#vCPUs*: the amount of CPU resources to use; (iii) *#vGPUs*: the amount of GPU resource to use. (Section 3.2 details the resource model). Then, the second core algorithm of ESG, *ESG_Dispatch*, assigns the task to an appropriate Invoker machine to run.

Note that ESG_1Q does not consider current resource availability constraints; as a result, ESG_Dispatch may not be able to find any Invokers having enough resource to run the task in a configuration found by ESG_1Q. ESG solves the issue by letting ESG_1Q output multiple top configurations, forming a *configuration priority queue*. ESG_Dispatch repeatedly dequeus the priority queue until it finds an Invoker that has enough resources to handle the configuration. If none of the configurations works, the scheduler records that AFW job queue in a *recheck list*, and moves on to the next AFW job queue. Each time it finishes processing a job queue, it tries again on the job queues in the *recheck list*; it may succeed now, as the states of the worker nodes have changed. If a queue stays in the *recheck list* too long (e.g., 3 rounds), it will be dispatched with the minimum configuration to ensure progress.

This two-step design is for two reasons: (i) finding optimal configurations takes some time and the states of the workers may have changed during that time; (ii) finding optimal configurations itself is already complicated, adding the dynamic changing machine status into the search process would further complicate the problem. The two-step design decouples the complexities and hence simplifies the solution. Meanwhile, by postponing the resource availability check to the second step, it is able to consider the current state of workers.

The two core algorithms both feature some novel designs, which help ESG address the two key challenges in adding sharable GPU into serverless computing, namely, the dramatically expanded resource assignment space and the increased complexity in handling dynamic changes in resource availability and function running times. We next first explain the resource and task models used in this work and then detail both algorithms.

3.2 Resource Model and Task Model

Resource model. A change to the resource model for heterogeneous serverless computing is the inclusion of GPU units. Modern GPUs are equipped with built-in mechanisms to support GPU partition and sharing. Multi-instance GPU (MIG) in NVIDIA GPU, for instance, allows GPUs to be securely partitioned into up to seven separate GPU Instances for CUDA applications. With MIG, each instance's processors have separate and isolated paths through the entire memory system, including the on-chip crossbar ports, L2 cache banks, memory controllers, and DRAM address buses that are assigned uniquely to an individual instance. We accordingly integrate the notion of vGPU into the serverless computing resource model. Here, each vGPU is equivalent to the minimum GPU partition of the sharing system (in our case, MIG). We assume each GPU is partitioned into the maximum number of MIG instances (7 for A100).

For a modern GPU-supported container, when it is launched, it can be set to use one or more vCPUs and one or more vGPUs. When it is set to use multiple vGPUs, it can use them by invoking multiple GPU kernels concurrently, one on each.

Current serverless platforms typically associate a certain amount of memory with one vCPU, which simplifies resource allocation: the platform can use vCPU as the allocation unit without explicitly allocating memory. GPU memory is associated with vGPU in a similar way, naturally enabled by MIG. We, however, do not associate vGPU with vCPU but make them separate resources for allocation. It is because there is no clear correlation between the amount of CPU usage and the amount of GPU usage in applications.

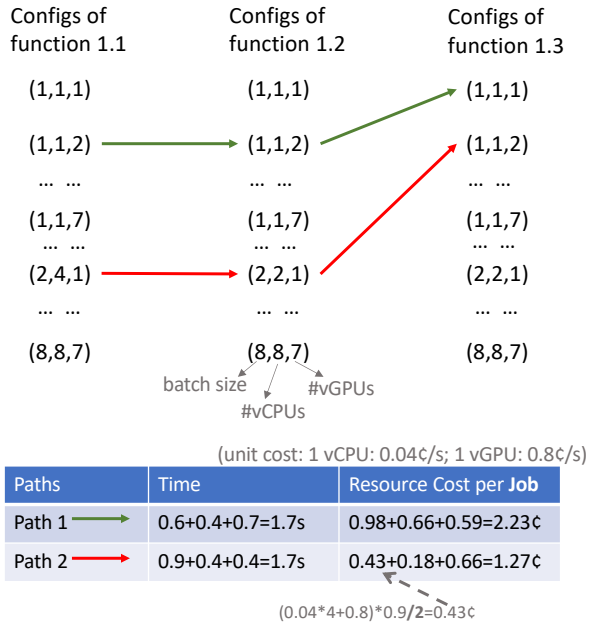
Task model. The applications targeted in this work are ML inference applications. (ML training is out of the scope.) There are two special extensions to the task model in heterogeneous serverless computing. (i) Each serverless function may now contain both CPU and GPU parts. (ii) For ML inferences, even though requests may come one by one, the inference function is often written in a way that it can accept a batch of requests and process them together at one invocation of the function. If the function is given multiple vGPUs, it automatically uses data parallel inferences by launching multiple GPU kernels with each processing one portion of the batch on one vGPU. We call the inference of one request a `job` and the set of jobs processed by an invocation of a serverless function a `task`.

3.3 ESG_1Q Algorithm

As Figure 2(d) shows, *ESG_1Q* tries to identify a good configuration for the jobs in an AFW queue so that the application can complete within the SLO latency with the minimum resource cost.

There are two complications: (i) speed-cost tension; a configuration that leads to faster execution tends to incur a higher cost. (ii) Inter-function dependence; the configuration selected for one function determines how much time is left for other functions in the application and hence affects how to best configure the other functions.

Our approach is to convert the problem into a path-finding problem in the configuration space of an application. The top part of Figure 3 illustrates the configuration space of a 3-function application. A path from the leftmost column to the rightmost column specifies the configurations for each of the three functions of the application. The red path in Figure 3, for instance, corresponds to a case where the first function processes two requests each time with



(a) Configuration space of an app and path costs

```
path_list = {}
new_path_list = {}
best_full_paths = {}  // the list of the best full paths
best_full_paths_maxCost = MAX_COST
foreach fun in functions of application A
  sort config_list(fun) in ascending order of resource cost
  foreach config in config_list(fun)
    foreach path in path_list
      new_path = path.append(config)
      // prune on time
      time_lowBound = getTimeLowBound(new_path)
      if (time_lowBound > TARGET_LATENCY)
        continue    // skip the path to avoid future consideration
      // prune on cost
      cost_lowBound = getCostLowBound(new_path)
      if (cost_lowBound > best_full_paths_maxCost)
        break
      new_path_list.insert(new_path)
      update best_full_paths & best_full_paths_maxCost
  path_list = new_path_list
  new_path_list = {}
if best_full_paths is empty
  setDefaultPaths(best_full_paths)
return best_full_paths
```

(b) Basic ESG_1Q algorithm in pseudo-code

Figure 3: (a) Top: Example of the configuration space of a three-function application and two configuration paths in the space. Bottom: the time and *per-job* resource costs of the two paths. (b) Basic ESG_1Q algorithm in pseudo-code.

four vCPUs and one vGPUs, the second function processes two requests with two vCPUs and one vGPU, and the third processes one request with one vCPU and two vGPUs.

Different paths result in different times and resource costs as exemplified by the table in Figure 3(a). The Controller can estimate the times with performance profiles of the functions and calculate the costs based on the unit costs of vCPU and vGPU and the running times.

With that formulation, the goal of ESG becomes finding the path in the configuration space of the application that meets the SLO latency and has the lowest resource cost. So each time, what ESG_1Q returns is not just a configuration good for the current function, but a sequence of configurations good for the whole application. It is important to note that unlike previous methods, Orion [43] and Aquatope [61], ESG calls ESG_1Q again when later functions in the application are to be scheduled, so that it can accommodate the dynamic resource changes and running time fluctuations.

A-Search with Dual-Bladed Pruning in ESG_1Q.* ESG_1Q uses A*-search as the basis for the path finding, and proposes two techniques, *Dual-bladed pruning* and *dominator-based SLO distribution*, to ensure the efficiency and scalability of the algorithm for serverless scheduling.

The A* search algorithm is an efficient and popular path finding and graph traversal method [31]. It efficiently finds the shortest path from a start node to a target node in a weighted graph. A* uses a heuristic to estimate the cost to reach the goal from each node, guiding the search towards the target more efficiently than algorithms like Dijkstra's, which only consider the actual cost from the start node. The total cost f(n) of a node n is calculated as f(n) = g(n) + h(n), where g(n) is the cost from the start node to n, and h(n) is the estimated cost (heuristic) from n to the goal. A* is a best-first search algorithm, meaning it explores a path that appears to be most promising by using the cost function f(n). It prioritizes paths that are expected to lead more quickly to the target. The algorithm maintains a priority queue (often implemented as a min-heap) of nodes to be explored, sorted by their f(n) value. A* is both complete and optimal, meaning it will always find a solution if one exists, and the solution will be the shortest possible path, provided that the heuristic function h(n) is admissible (it never overestimates the true cost) and consistent (monotonic).

ESG_1Q builds on A*, the optimality of which makes ESG scheduling optimality-guided. ESG_1Q meanwhile improves the path finding with *dual-bladed pruning*. Figure 3 (b) outlines the basic idea. It prunes the search space based on both running time and resource usage, efficiently avoiding exploration of unnecessary subspaces. The pruning effectively estimates the lower and upper bounds of the costs of a path. When a partial path p forms, ESG_1Q calculates three bounds, *tLow*, *rscLow*, and *rscFastest*, and uses them for pruning, as follows:

- *tLow*: the lower bound of the time cost of all paths prefixed by p. ESG_1Q estimates it by summing the time of the functions in p (obtained from the profiles) and the minimum time of each function (among all its possible configurations) not covered by p. It is used in time-based pruning (function getTimeLowBound in Figure 3 (b)).
- *rscLow*: the lower bound of the resource usage of all paths prefixed by p. ESG_1Q estimates it by adding the resource usage of the functions in p (obtained from the profiles) and

the minimum resource usage of each function (among all its possible configurations) not covered by p. It is used in the cost-based pruning (function getCostLowBound in Figure 3 (b)).

- $rscFastest$: the summation of the resource consumed by p and the resource consumed by other functions when they run the fastest. It is used in updating best_full_paths_maxCost in Figure 3 (b) to tighten the bound for cost-based pruning.

Note that for ease of understanding, Figure 3 (b) shows only the basic design of ESG_1Q; Omitted details (e.g., the use of priority list for best-first search) are documented in Appendix B.

Dominator-based SLO Distribution for Scalability. Even with Dual-blade pruning, when dealing with lengthy call sequences within an application, the algorithm's execution time can still be exceedingly long. To address this issue, we introduce a strategy named *dominator-based SLO distribution*. It uses *stage grouping* to divide the functions of the application into several function groups, uses *SLO distribution* to assign a specific SLO latency to each group, and then applies the ESG_1Q search algorithm to each individual group. To ensure maximal quality, the groups are maximized under the constraints of tolerable schedule latency. This strategy provides a way to strike a good balance between the scheduling scalability and the benefits of considering inter-function relations in scheduling.

For this approach to work, we need to determine how to group functions and assign individual SLOs to these groups. This solution should be adaptable to both linear pipelines and more complex DAGs with splits and joins.

Our solution is a *reduction-based hierarchical* method. It is based on an observation that the DAGs in serverless applications are usually hierarchically reducible, as Figure 4 illustrates. The design is inspired by dominator-based code analysis in compilers [14]. The algorithm includes four steps.

First, it creates the dominator tree of the DAG following traditional compiler-based code analysis [14]. In a dominator tree, each edge indicates an immediate domination relation in the DAG[2], as illustrated in Figure 4. The data structure offers the basis for an ordered traversal and reduction in processing.

Second, it labels each node in the dominator tree with the *average normalized length (ANL)* of its function (say f_i), $average_c(l_{f_i}(c))$, where c is a configuration, $l_{f_i}(c)$ is the normalized length of the function f_i in configuration c, calculated as $t_{f_i}(c)/\sum_j t_{f_j}(c)$; $t_{f_j}(c)$ is the execution time of function f_i in configuration c, read from the performance profile.

Third, it traverses the dominator tree in a post-order (children before parents). At each node (say x), if it has more than one child, it calls subroutine *reduce(x)* to first reduce and reorganize its descendants. The *reduce* operation is to combine its branches into one node, as illustrated in Figure 4 (c). The ANL of that new node is the maximum of the sums of the ANL of all the branches. After that, the algorithm checks if the parent of x has more than one child and if so, it calls the subroutine *slo_group(x)* to partition x and its descendants into one or more groups. Note that thanks to *reduce*, the descendants of x are guaranteed to form a single list. The partition simply groups consecutive g nodes into one group, where g is the maximum group size (Section 5.4 reports how g affects performance). An exception is the nodes generated by reduction, each of which stays as an individual node to prevent their subsumed groups' sizes from being bloated. This process continues until the algorithm reaches the root. Throughout the process, the algorithm records the reduction process for the next step to use.

Finally, the algorithm assigns SLO latency to each of the groups. This process reverses the reduction process. It starts from the final form of the reduced dominator tree (which is a list) and partitions the end-to-end SLO latency proportionally based on the ANLs of its groups. It then calls subroutine *slo_assign(x)* for each reduced node in the current list to partition the SLO latency of x and assigns the partitions to the nodes that x subsumes. This process continues until every group gets its SLO quota.

[2]A dominates B if all paths from the root to B must first reach A; an immediate dominator is the closest dominator except the node itself [14].

3.4 ESG_Dispatch: Mapping to Worker Nodes

After ESG_1Q, **ESG_Dispatch** maps the current group of jobs to an Invoker node. As we introduced in Section 2, the OpenWhisk scheduling always chooses the home-invoker first; if not feasible, it tries other worker nodes. Our algorithm chooses the home-invoker for the first function in the workflow. For other functions, it would try to choose the invoker that runs its predecessor function in the workflow. This locality-sensitive treatment is possible thanks to our introduced AFW queues. It helps reduce data transfer, as communications on the same node can use local file systems rather than remote storage. This consistent policy is beneficial for getting warm starts. If the home-invoker or predecessor-invoker does not have enough available resources, the algorithm will try other warm Invokers. If it fails, it will check other cold invokers and choose the one with the most available resources.

4 METHODOLOGY FOR EVALUATION

To evaluate the efficacy of the scheduling algorithm, we create a framework that can emulate various serverless workloads and scenarios. The emulations are based on actual performance of the serverless functions measured on actual machines in various configurations (batch size, CPU and GPU resource allocations). The machine is as specified in Table 2. To accommodate the impact of other runtime factors on the performance, the emulations add Gaussian noises to the performance. The emulation is equipped with a workload generator, which generates workloads by sampling the set of serverless functions randomly based on a specified arrival rate. The set of workloads we considered in the evaluation are further detailed in Sec 4.1. The hardware resource of the considered testbed in the emulations consists of 16 nodes, with each equipped with 16 vCPUs and 1 GPU (up to 7 vGPUs by MIG). The scheduler and job dispatching implementation is based on the controller in OpenWhisk [1].

We use proxy threads to monitor the function call intervals, predict subsequent invocations, and preemptively warm up instances. Studies have shown that pre-warming can help reduce delays caused by cold starts. There have been several methods proposed before, with some using ML models [61] and others [42, 43, 54, 58] using a histogram-based policy to adjust container keep-alive times. We use a lightweight method for prewarming. It uses

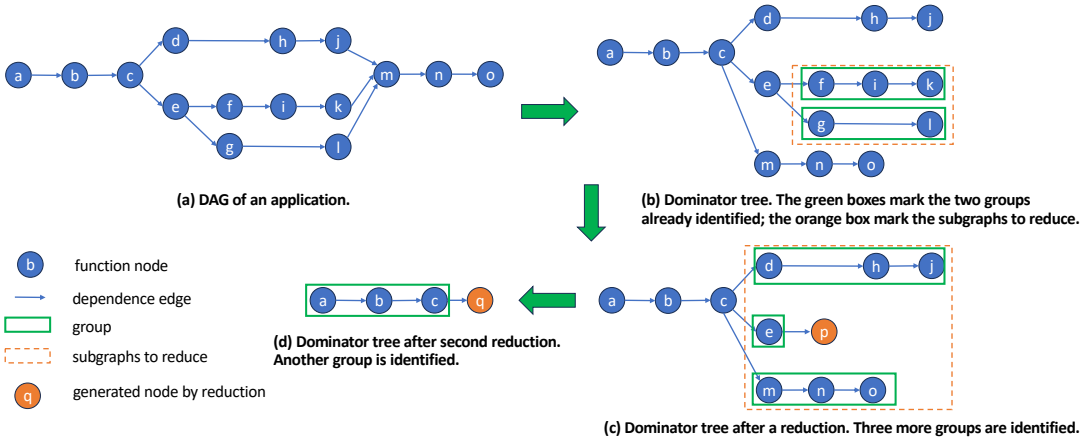


Figure 4: Illustration of dominator-based SLO distribution.

Exponential Weighted Moving Average (EWMA) [6] to predict the invocation intervals of functions and pre-warms the function instances accordingly. After pre-warming, ESG uses the same keep-alive policy as OpenWhisk, to keep the instance alive for 10 minutes.

Table 2: Experimental testbed configuration

CPU device	AMD EPYC 7302P 16-Core Processor
CPU Mhz	1499.866
CPU memory	128GB DDR4 3200MHz ECC DRAM
GPU device	NVIDIA A100 80GB
GPU memory	80GB
MIG instances	Up to 7 MIGs@10GB

4.1 Applications

The workloads in the experiments are series of calls to four applications, with each consisting of a sequence of DNN inferences. Table 3 reports the source, inputs, DNN models, execution time in the minimum configuration (1vCPU, 1vGPU, batch size=1), and cold start time of the functions. The four applications are detailed as follows:

- **Image classification:** It detects and classifies objects, important for autonomous driving [29] and other domains. Its workflow is to use super-resolution [39] first to enhance the clarity of an image, and then use segmentation [5, 20] followed by classification [12, 32] to identify the objects.
- **Depth recognition:** This application measures the distance of an object from a camera, which is essential to 3D scene reconstruction and augmented reality [37]. Its workflow uses deblur [4] followed by super-resolution first to enhance the image and then recognize image depth with another DNN [49].
- **Background elimination:** This application eliminates unnecessary and unwanted items and objects from images [18]. Its workflow is super-resolution followed by deblur to enhance the image clarity and then uses background removal DNN [47] to eliminate the background.

Table 3: Serverless Functions

Function name	Execution Time (ms)	Cold start time (ms)	Input image size (MB)	Model
Super resolution [39]	86	3503	2.7	SRGAN
Segmentation [5, 20]	293	16510	2.5	deeplabv3_resnet50
Deblur [4]	319	22343	1.1	DeblurGAN
Classification [12, 32]	147	18299	0.147	ResNet50
Background removal [47]	1047	3729	2.5	U^2 Net
Depth recognition [49]	828	16479	0.648	MiDaS

- **Expanded image classification:** This is a more advanced image classification application with a longer workflow: deblur, followed by superresolution, background removal, segmentation, and classification.

For the workload setting, we examined the traces published by Azure [54] to determine job arrival rates. We get the job arrival rates of every minute from the Azure traces, based on which we derived three situations for our DNN applications respectively with **heavy**, **normal**, and **light** workloads. In each workload, one of the four DNN applications is randomly picked to get invoked in each time interval. The length of a job arrival interval is selected randomly in ranges [10–16.8ms], [20–33.6ms], and [40–67.2ms] respectively in the three situations. Figure 5 shows the distribution of the job arrival intervals in the three situations.

We tested three SLOs. Let L be the time needed by the application to complete its entire workflow when it runs alone with the minimum configuration. (i) In the **strict** setting, a SLO hit occurs when the application completes within $0.8 \times L$. (ii) In the **moderate** setting, a SLO hit happens when the application completes within $1 \times L$. (iii) In the **relaxed** setting, a SLO hit happens when the application completes within $1.2 \times L$.

Those three levels of requirements correspond to users' possible expectations of service: strict for the light case, moderate for the normal case, and relaxed for the heavy case, denoted as **strict-light**, **moderate-normal**, and **relaxed-heavy** workloads in the evaluation section.

Following AWS EC2 pricing [58], we set the price of a vCPU to 0.034$/hour. Based on the pricing of an entire GPU on AWS, we divide it by # of vGPUs and set the price of a vGPU to 0.67$/hour.

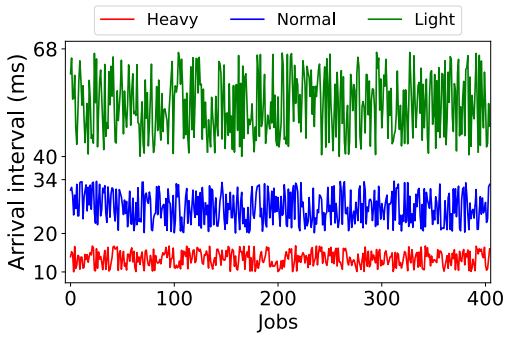


Figure 5: Job arrival intervals used in the evaluation part for different workload settings.

4.2 Comparison Counterparts

We evaluate our ESG algorithm by comparing it with four state-of-the-art scheduling algorithms, including **INFless** [58] and **FaST-GShare** [30], the latest algorithms for sharable GPU-based serverless ML, as well as the **best-first search** algorithm in Orion [43] and the **Bayesian Optimization-based scheduling** in Aquatope [61]. The original Orion [43] and Aquatope [61] do not support GPU sharing, but their scheduling algorithms represent the latest search-based scheduling and model-based scheduling, respectively. For comparison, we extend both with GPU sharing support, as explained below.

InFless: InFless schedules jobs by enumerating the configurations for each function without considering the inter-function relations. In worker node selection, a resource efficiency metric is used to maximize the throughput while reducing resource fragmentation. InFless provides no method for distributing an application's SLO to its functions. Our experiment follows a prior work [36] to do the distribution based on the average service times of the functions.

FaST-Gshare: This work uses FaST-Manager to manage spatio-temporal resources for GPU multiplexing. It also employs an enumeration-based scheduling algorithm which enumerates the configurations based on throughput performance metrics. Its node selection tries to minimize GPU resource fragmentation. It offers no method for distributing an application's SLO either. We apply the same method as in INFless.

Aquatope: Aquatope relies on an offline training process, in which the application of interest is profiled in many sample executions based on Bayesian Optimization (BO), through which it builds up a performance model and learns about the statistically good configurations for every stage in the application (encoded in an *acquisition function*). In deployment, it uses the learned best configurations for the application. Specifically, the training process starts with 100 bootstrapping samples, iterates 50 rounds (we sample five configurations in each round), and selects the best configuration. The nature of its reliance on offline training makes it unable to adapt to dynamic workload changes, as shown in the next section.

Orion: Orion creates a performance model to address runtime variations, consolidates parallel invocations into a single virtual machine (VM), and implementes instance pre-warming to eliminate cold starts. Its scheduling uses *best-first search*, which creates a priority queue, in which all new states are added. Adding vGPU into the algorithm, we expand its state definition to a vector of (batch size, #vCPUs, and #vGPUs), one for each stage. The algorithm examines possible states, with each new state increasing the current state in one dimension of the configuration vector, and the start state S_0 has the minimum values for every stage function. The scheduling method decides the schedule for all the stages of an application at the invocation of the first stage; no dynamic adaptation between stages. As in the original work, P95 latency is used as the search goal. The configuration with the closest latency to the SLO is returned when the search exceeds a cut-off time (e.g., 100ms) before reaching the goal.

The GPU-sharing and batching policy can improve resource utilization, as shown in Section 5.5. So, to evaluate the effectiveness of the scheduling algorithm, in our comparisons, we enable the same GPU-sharing and batching for all the scheduling algorithms; the same data locality and pre-warming policy proposed in this work are also used; the only difference is the scheduling algorithm. INFless and FaST-GShare do not follow the data locality policy but their resource fragmentation minimization policy.

5 EVALUATION

Our evaluation examines (i) the overall effectiveness of ESG in maximizing SLO hit rates while minimizing the cost; (ii) the reasons for the benefits of ESG over the state-of-the-art scheduling algorithms (INFless [58], FaST-GShare [30], Orion [43] and Aquatope [61]); (iii) the overhead and sensitivity analysis of ESG.

5.1 End-to-End Performance

Figure 6 shows the average SLO hit rates and total normalized cost (ESG is 1) for all applications across three situations. In all three scenarios, ESG consistently shows a high SLO hit rate. Its benefit is especially pronounced in the strict-light scenario: Its hit rate is 46%-80% higher than BO and Orion, and 36%-61% higher than INFless and Fast-GShare. It is noteworthy that ESG achieves the significantly higher SLO hit rates with similar or much less resource cost compared to other methods, as Figure 6 shows. Detailed SLO hit rates and cost of each application are shown in Figure 8. ESG consistently achieves the highest SLO hit rate at a lower cost, whereas INFless consumes the most resources.

For a more detailed view, Figure 7 shows the end-to-end latencies of each of the four applications in the relaxed-heavy setting. ESG consistently achieves latencies below but close to the SLO latency. The configurations found by other methods cause the jobs to either run too slow (e.g., FaST-GShare) or to use more resources (e.g., INFless) than necessary at the expense of larger cost or poor performance of other applications.

For instance, INFless selects configurations that yield lower latencies for applications such as "image classification", "depth recognition", and "background elimination" compared to other methods. This is due to its resource efficiency metric in scheduling, which reduces resource fragmentation and increases system throughput, thereby preferring to utilize all remaining resources in one invoker. However, this approach of allocating excessive resources, as confirmed by the highest resource costs shown in Figure 8, leads to prolonged waiting times and increased latency for the "expanded

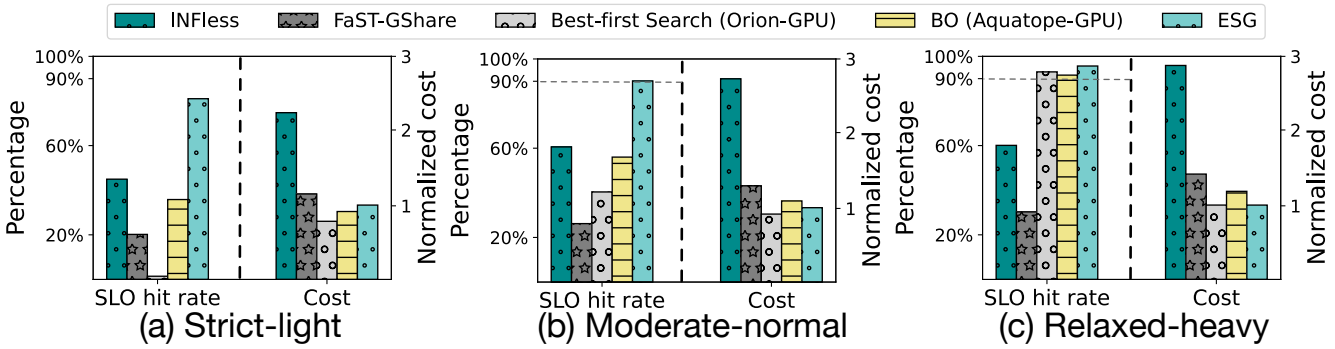


Figure 6: The average SLO hit rate and the cost (normalized to ESG cost) under different SLO and workload settings. The left y-axis is for the SLO hit rate and the higher is better. The right y-axis is for the cost and the lower is better.

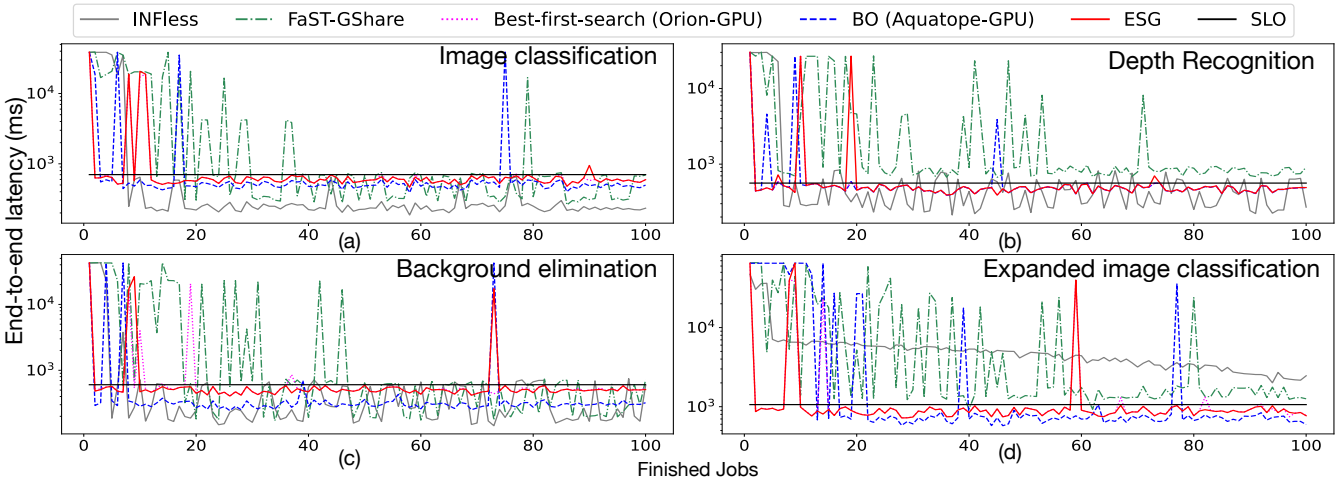


Figure 7: End-to-end latency for each application in relaxed-heavy setting.

image classification" application. This application is particularly affected due to its extensive pipeline, which involves more pipeline stages awaiting resources.

5.2 Detailed Analysis

In this section, we provide a detailed analysis to examine the reasons for the benefits of ESG over other methods.

Compared to INFless and FaST-GShare: INFless and FaST-GShare distribute the end-to-end SLO to different stages based on their average service times, without considering their inter-dependencies. Consequently, if certain early stages experience delays due to data transfer overhead and cold start latency, the later stages do not adjust their SLO settings, resulting in prolonged execution time, especially when the application has multiple functions. As demonstrated in Figure 7(d), FaST-GShare and INFless always yield the largest latency. Furthermore, in selecting worker nodes, they prefer reducing resource fragmentation rather than focusing on data locality and data transfer latency. This results in even lower SLO hit rates, a fact that is evidenced by the numerous strikes seen in the FaST-GShare curve of Figure 7.

Compared to Orion: Orion is a search-based method. It faces a trade-off between the search time and the quality of the search result. Figure 9 shows the tradeoff in the strict-light setting. In this setting, Orion can find quite good configurations. The blue curve in Figure 9 shows the hit rates of those configurations when search overhead is not counted in. But when the search time is counted in, the hit rates drop dramatically, as the green curve in Figure 9 shows. Moreover, because Orion decides the configurations for all functions in an application when scheduling the first function and does not adjust the configurations of later stages, the configurations are often low in quality or do not even apply. It is especially common when the resource availability changes significantly along time. Table 4 shows the percentage of times when the configurations fail to apply to a function because the batch size in the configuration is even greater than the number of jobs in the queue of that function when it is time to be scheduled. In the moderate-normal and relaxed-heavy settings, because resource availability changes substantially, the percentage is as much as 27–52%.

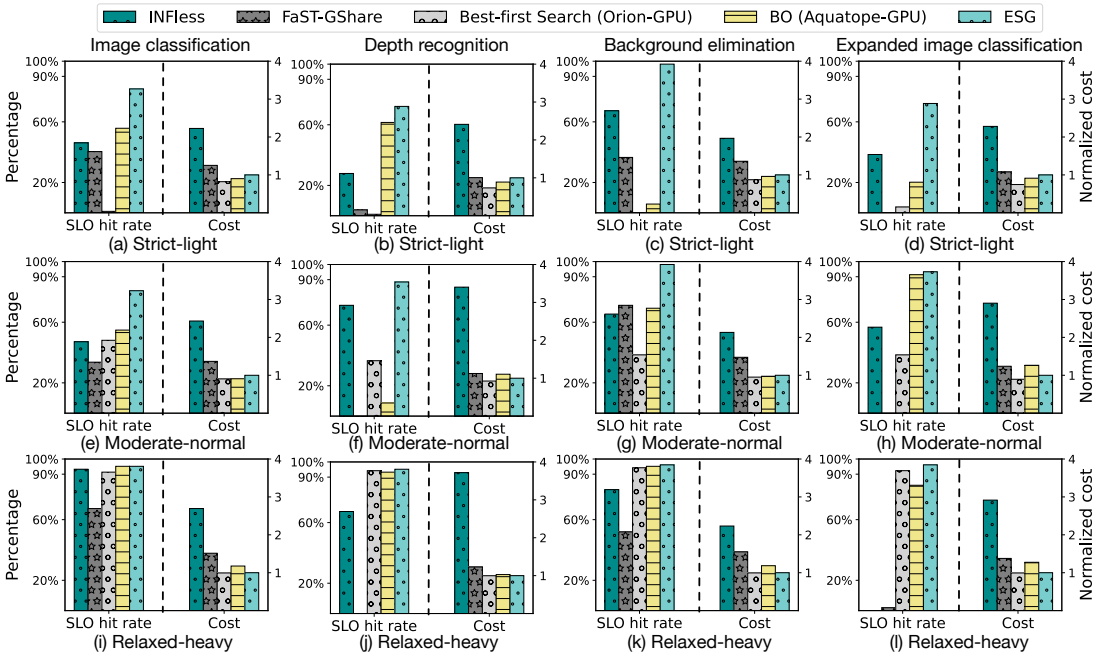


Figure 8: SLO hit rates and cost for each application in three different workload settings.

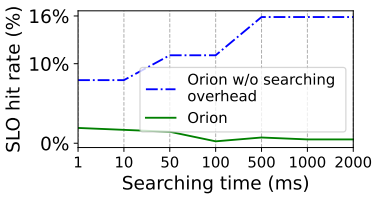


Figure 9: The effect of search time of Orion on the SLO hit rates (strict-light setting).

ESG overperforms Orion because (i) ESG finds better configurations much faster, thanks to its dual-bladed pruning and dominator-based SLO distribution (overhead analysis shown in the next subsection); (ii) ESG adapts the configurations for every function in a workflow. As a result, with a smaller overhead, ESG produces configurations that meet the SLO and demand fewer computing resources. The lower computing resource demand gives multi-fold benefits. It not only lowers the cost, but also makes the function more likely to fit into the available resource of the predecessor worker node, which in turn leads to better data locality, less communication overhead, and fewer cold starts, which all contribute to the higher SLO hits.

Compared to Aquatope: Aquatope relies on a statistical model trained with offline traces. It has negligible scheduling overhead. However, ESG surpasses Aquatope because of the better quality of the configurations given by ESG. As shown in Figure 5, real workloads and resource conditions continually change. Being a method based on an offline training model, the BO method cannot adapt to dynamic changes. It assumes that configurations remain

Table 4: Pre-planned scheduling miss rate

2*System setting	Configuration miss rate	
	Best-first search (Orion)	BO (Aquatope)
Strict-light	9.6%	85.5%
Moderate-normal	27.32%	59.85%
Relaxed-heavy	51.68%	58.72

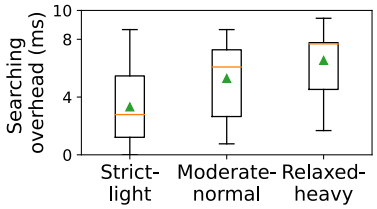


Figure 10: Scheduling overhead distribution of ESG (function group size is 3).

unchanged unless functions or inputs are modified, but this static scheduling configuration proves inadequate in an ever-changing real-world environment where future job and worker node statuses are uncertain. It is confirmed by the rightmost column in Table 4, which shows that 59–86% of the configurations preset by the BO method do not apply in actual executions because the actual queue length is smaller than the batch size in the configuration.

5.3 Overhead Analysis

Figure 10 reports the scheduling overhead distribution of ESG in the three settings (using the default function group size 3), with the green triangle indicating the average. The searching overhead

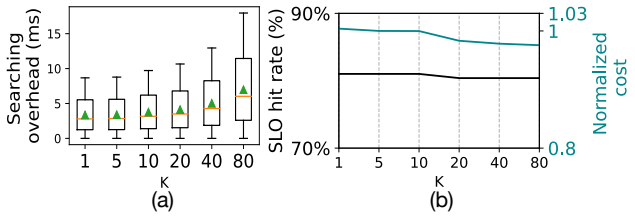


Figure 11: Sensitive study of (K) in strict-light setting, the cost of K=5 is set to be 1.

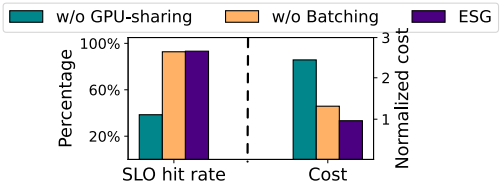


Figure 12: The ablation study in relaxed-heavy setting.

increases with more relaxed SLO settings. It is because in relaxed SLO settings, more configurations can satisfy the SLO, resulting in fewer configurations being pruned during the search process. But overall, the search overhead is less than 10ms. In comparison, the time taken by a brute-force search would be orders of magnitude higher. In the case where each function has 256 configurations, the search time is 7258ms.

5.4 Sensitive study

In ESG, there are two parameters, the maximal size of a function group and the number of solutions in the configuration priority queue (K). The default maximal group size is set to 3 because when the size increases to 4, the search time jumps to 1201ms (for 256 configurations per function) due to the exponential growth of the configuration space. Regarding the impact of K value, Figure 11 reports our observations. As K increases from 1 to 80, the average search overhead increases from 3ms to 8ms, the latency remains similar, and the cost decreases slightly. The default K is set to 5 in ESG.

5.5 Impact of GPU-sharing and Batching

Through our ablation study, we assess how the GPU-sharing and batching strategies enhance GPU resource efficiency. We individually removed either the GPU-sharing or batching strategy from ESG and contrasted the results with the original ESG. We set a heavy workload in this experiment specifically to underline the effects of the batching strategy. The results indicate that both strategies boost the usage of GPU resources, as evident in Figure 12.

Without the GPU-sharing strategy, the waiting time is substantially prolonged compared to ESG. This is because jobs are queued, waiting for a GPU (currently in use) to free up. Consequently, the data-locality strategy may falter, leading to substantial data transfer costs and worse SLO hit rates. The batching strategy is crucial in conserving cost, as shown in Figure 12. The batching policy will delay the execution time; thus, no-batching policy will not decrease the SLO hit rates.

6 RELATED WORK

Heterogeneous serverless computing: Recent studies on GPU-based serverless computing focused on CPU-GPU data transfer [33, 60] and cold start overhead [52, 53]. Some prior works [38, 46] studied the scheduling tasks on GPUs but regarded an entire GPU as the minimal computing unit. Other works [22, 30, 58] proposed the GPU-sharing scheduling for inference applications but neglected the inter-function relations of the DAG workflows. There have been efforts to extend serverless computing support to heterogeneous hardware, including FPGAs and Nvidia DPUs [25, 56]. Additionally, Dgsf [28] proposed the disaggregated GPU for serverless computing that is orthogonal to our work.

Serverless resource management: Several works [21, 35, 43, 48, 55, 58, 61] have studied the resource management problem in serverless computing. Orion [43] and INFless [58] are search-based scheduling algorithm. SIMPPO [48] used reinforcement learning for serverless resource management. Aquatope [61] is a Bayesian Optimization based scheduling algorithm, which builds upon IceBreaker [51] and CLITE [45] and extends BO in new ways than what was previously done in other BO-inspired solutions like SATORI [50] or Ribbon [40], and inspired the OLPart [21].

Optimization of data transfer: Researchers realize the data transfer between workers and remote database lead to unnecessary latency. FaaSFlow [41] proposed to partition the workflow into sub-graphs and schedule these functions in one invoker to avoid data transfer. Nightcore [34] proposed the internal function calls and low-latency message channels, and efficient threading for I/O to reduce the data transfer latency. Sonic [42] proposed the hybrid data passing methods, which are direct-message passing and remote storage to reduce the data transfer latency. Palette [13] used the "colors" to place successive invocations related to each other on the same executing node. As shown in previous sections, due to the lack of effective methods to handle large schedule space, prior work left much potential for sharable GPU locked for ML on serverless platforms.

7 CONCLUSION

This study has proposed and evaluated a new algorithm ESG to effectively schedule ML workloads on serverless platforms with sharable GPUs. The search algorithm employs an *optimality-guided adaptive* method by combining A*-search and a novel *dual-blade pruning* to effectively prune the scheduling space without compromising the quality. Its *dominator-based SLO distribution* offers a way to keep the algorithm scalable. Across a diverse set of real-world serverless applications, ESG gives the highest SLO hit rates, while significantly reducing the cost.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grants No. CNS-2312207, CNS-2107068, and CMMI-2246671. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Apache OpenWhisk. How OpenWhisk works. https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works.

[2] AWS Lambda. https://aws.amazon.com/lambda/.

[3] Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview.

[4] DeblurGAN. https://github.com/pablodz/DeblurGAN.

[5] DEEPLABV3. https://pytorch.org/hub/pytorch_vision_deeplabv3_resnet101/.

[6] Exponentially Weighted Moving Average. https://www.sciencedirect.com/topics/computer-science/exponentially-weighted-moving-average.

[7] Google Cloud Functions. https://cloud.google.com/functions.

[8] Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. https://knative.dev/docs/.

[9] NVIDIA 2020, Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[10] NVIDIA, Multi-Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html.

[11] OpenWhisk. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[12] ResNet50. https://pytorch.org/hub/nvidia_deeplearningexamples_resnet50/.

[13] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 365–380.

[14] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley.

[15] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 923–935.

[16] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[17] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*. 263–274.

[18] David Bau, Hendrik Strobelt, William Peebles, Jonas Wulff, Bolei Zhou, Jun-Yan Zhu, and Antonio Torralba. 2020. Semantic photo manipulation with a generative image prior. *arXiv preprint arXiv:2005.07727* (2020).

[19] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

[20] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. 2017. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587* (2017).

[21] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. 2023. OLPart: Online Learning based Resource Partitioning for Colocating Multiple Latency-Critical Jobs on Commodity Computers. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 347–364.

[22] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. 2022. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proceedings of Machine Learning and Systems* 4 (2022), 20–32.

[23] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216.

[24] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[25] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 797–813.

[26] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 45–59.

[27] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110* (2020).

[28] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. 2022. DGSF: Disaggregated GPUs for Serverless Functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 739–750.

[29] Hironobu Fujiyoshi, Tsubasa Hirakawa, and Takayoshi Yamashita. 2019. Deep learning-based image recognition for autonomous driving. *IATSS research* 43, 4 (2019), 244–252.

[30] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. 2023. FaST-GShare: Enabling Efficient Spatio-Temporal GPU Sharing in Serverless Computing for Deep Learning Inference. *arXiv preprint arXiv:2309.00558* (2023).

[31] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[33] Sungho Hong. 2022. *GPU-enabled Functional-as-a-Service*. Ph. D. Dissertation. Arizona State University.

[34] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[35] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2022. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*. 289–305.

[36] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[37] G Ajay Kumar, Jin Hee Lee, Jongrak Hwang, Jaehyeong Park, Sung Hoon Youn, and Soon Kwon. 2020. LiDAR and camera fusion approach for object distance estimation in self-driving vehicles. *Symmetry* 12, 2 (2020), 324.

[38] Vincent Lannurien, Laurent d'Orazio, Olivier Barais, Esther Bernard, Olivier Weppe, Laurent Beaulieu, Amine Kacete, Stéphane Paquelet, and Jalil Boukhobza. 2023. HeROfake: Heterogeneous Resources Orchestration in a Serverless Cloud–An Application to Deepfake Detection. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 154–165.

[39] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. 2017. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4681–4690.

[40] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. 2021. Ribbon: cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[41] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 782–796.

[42] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware data passing for chained serverless applications. In *USENIX Annual Technical Conference (USENIX ATC)*.

[43] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[44] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–28.

[45] Tirthak Patel and Devesh Tiwari. 2020. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 193–206.

[46] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez. 2022. Kernel-as-a-Service: A Serverless Interface to GPUs. *arXiv preprint arXiv:2212.08146* (2022).

[47] Xuebin Qin, Zichen Zhang, Chenyang Huang, Masood Dehghan, Osmar Zaiane, and Martin Jagersand. 2020. U2-Net: Going Deeper with Nested U-Structure for Salient Object Detection. *Pattern Recognition* 106, 107404.

[48] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2022. SIMPPO: a scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing*. 306–322.

[49] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer. arXiv:1907.01341

[50] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. Satori: efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 292–305.

[51] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.
[52] Justin San Juan and Bernard Wong. 2023. Reducing the Cost of GPU Cold Starts in Serverless Deep Learning Inference Serving. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 225–230.
[53] Justin David Quitalig San Juan. 2023. *Flashpoint: A Low-latency Serverless Platform for Deep Learning Inference Serving*. Master's thesis. University of Waterloo.
[54] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.
[55] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. 2023. Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 301–314.
[56] Jessica Vandebon, Jose GF Coutinho, and Wayne Luk. 2021. Scheduling Hardware-Accelerated Cloud Functions. *Journal of Signal Processing Systems* 93 (2021), 1419–1431.
[57] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.
[58] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 768–781.
[59] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. 2017. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 1–13.
[60] Ming Zhao, Kritshekhar Jha, and Sungho Hong. 2023. GPU-enabled Function-as-a-Service for Machine Learning Inference. *arXiv preprint arXiv:2303.05601* (2023).
[61] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2023. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1–14.

A PROBLEM FORMAL DEFINITION

For clarity, we provide a formal definition of the scheduling problem as follows.

Assumptions:

- One serverless function invocation uses at most one worker node.
- All worker nodes have the same hardware resources (we made this assumption for the explanation simplicity, our algorithm still works with heterogeneous hardware resources).
- The serverless functions are in a form that can accept a single job or a batch of jobs.

Given: A set of jobs $B = \{e_i \mid I \geq i > 0\}$ with each job as an invocation of one of the serverless functions $\{f_j \mid J \geq j > 0\}$; each job in B belongs to one and only one of the applications $A = \{a_m \mid M \geq m > 0\}$ and one application consists of one or more jobs; each application a_m has a tolerable latency upper limit d_m; a set of workers $\{w_k \mid K \geq k > 0\}$ with each worker being equipped with R_C CPU resource units and R_G GPU resource units;

Objective: Produce a set of schedule configurations $C = \{c_l = (b_l, r_l, p_l, q_l) \mid L \geq l > 0\}$, where each configuration c_l represents that at time q_l, a set of jobs b_l are assigned to the worker p_l to run with r_l=(u_{c_l}, u_{g_l}) resources (u_c for CPU resource units, u_g for GPU resource units), such that:

- $argmin_C \sum(\alpha u_{c_l} + \beta u_{g_l}), 1 \geq \alpha \geq 0, \beta = 1 - \alpha$.

Constraints:

- $\sum h_m > \gamma$, where h_m=1 if the end-to-end time of application m, denoted as t_{a_m}, is no greater than d_m, and 0 otherwise; $t_{a_m} = \sum t_{b_i}$ where t_{b_i} is the end-to-end time of job b_i, that is, the time from its invocation to its completion.
- $\cup_l^L b_l = \cup_i^I e_i$ (every job is scheduled)
- $b_i \cap b_j = \emptyset, \quad \forall l \geq i, j > 0$ and $i \neq j$ (each job belongs to only one set)
- $\sum_{l=0}^{L} u_{c_l} \mathbf{1}(b_l, k, t) \leq R_C, \quad \forall t, \forall K \geq k > 0$, where, $\mathbf{1}(b_l, k, t)$ is 1 if b_l is active at time t on worker k, and 0 otherwise. (within total CPU resource)
- $\sum_{l=0}^{L} u_{g_l} \mathbf{1}(b_l, k, t) \leq R_G, \quad \forall t, \forall K \geq k > 0$, where, $\mathbf{1}(b_l, k, t)$ is 1 if b_l is active at time t on worker k, and 0 otherwise. (within total GPU resource)

B ESG_1Q ALGORITHM

Received 25 January 2024; accepted 25 March 2024

Algorithm 1: ESG_1Q

Input: i ← the current stage
Input: ***endStage*** ← the final stage of the function sequence
Input: ***w*** ← the longest waiting time;
Input: ***q*** ← the time quota, which is got by the dominator-based-distribution method, of this sequence
Input: ***The target latency*** (G_{SLO}) ← ***(SLO - w)*** × ***q***
Input: ***ConfigLists[j]*** ← the profiles of function j sorted in increasing latency
Output: ***configPQ***={}, the final feasible configurations and sorting by the resource cost
Data: ***minRSC*** ← a sorted list to maintain K best paths, which is used for pruning on the resource usage. (***K*** is the solution number we set)
Data: ***Paths***={}, the feasible paths until now and sorting by the resources cost.

```
for each config in ConfigLists[i] do
    calculate tLow, rscLow, recFastest;
    if tLow >= G_SLO then
        break ;                    /* Pruning on time */
    end
    if rscLow >= minRSC[K-1] then
        continue ;   /* Pruning on cost, minRSC[k-1]
         is the best_full_paths_maxCost in the
         main paper */
    else
        Remove minRSC[K-1] and insert rscFastest into
         minRSC;
        Add the config into Paths;
    end
end
while i+1 <= endStage do
    Reset minRSC = [];
    while Paths is not empty do
        Dequeue one path from Paths;
        for each config in ConfigLists[i+1] do
            newPath ← Extend the path by appending the
             config;
            calculate tLow, rscLow, recFastest;
            if tLow >= G_SLO then
                break;
            end
            if rscLow >= minRSC[K-1] then
                continue;
            end
            if i+1 == endStage then
                Insert the newPath into the configPQ;
            end
            Remove minRSC[K-1], insert rscFastest into
             minRSC;
            Add newPath into Paths;
        end
    end
    i = i + 1;
end
Return configPQ;
```