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

Xinning Hui, Zhishan Guo, Xipeng Shen (North Carolina State University) Yuanchao Xu, (University of California, Santa Cruz)

HPDC' 2024 Jun

Presenter: Mausam Basnet, University of North Texas

Date: 2024-11-01

Introduction

Growing Interest in ML Inference on Serverless Platforms:

- Ease of programming, maintenance, autoscaling, and pay-as-you-go billing

Current Gaps:

- Major platforms (AWS Lambda, Google Cloud Functions, Azure Functions) remain CPU-centric
- Lacking native GPU support limiting ML efficiency on serverless setups.

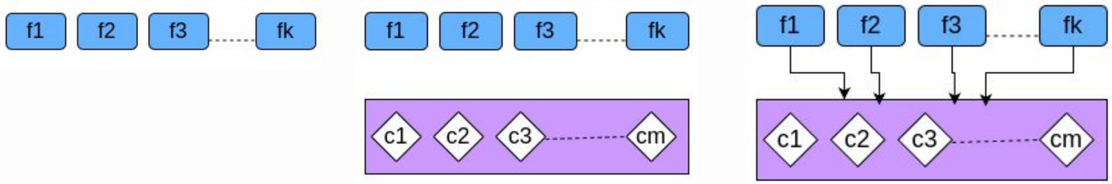
GPU Need for ML:

- ML tasks are compute-intensive; GPU support can enhance throughput by leveraging GPU parallelism

Research Efforts: Studies on GPU sharing using NVIDIA MPS or increasing throughput by batching.

Challenge: The dramatically expanded search space for scheduling

A key step in serverless scheduling is configuring resources for each function to meet SLOs with minimal resource use.



If no GPU sharing

Only configuration is number of vCPUs

So, Configuration space = m^k

Fig: Functions and configurations

If sharing GPU:

The configuration would be:
batch size, number of vCPUs, number of vGPUs

Configuration space = (m^k)^3

** for m = 5 and k=7 : the space increases from 78K options to 476 trillions*

Challenge: Complexity in handling performance variations

Why Performance Variations Occur in Serverless ML Inference ?

- Tasks arrive at varying rates, sometimes in large "bursts."
- Multiple functions compete for the same GPU, impacting availability.
- Resource availability changes constantly based on other ongoing tasks.

Challenges in Managing These Variations

- Schedulers need to frequently re-evaluate resources based on real-time function performance.
- Achieving consistent SLO compliance is difficult with unpredictable execution times.
- Adaptive Scheduling Requirement:
  - Must adjust configurations on the fly.
  - Balance SLO targets with changing GPU availability and function demands.

Serverless computing architecture

OpenWhisk: An Open-Source Serverless Platform

1. Event-Driven Execution:
   - Executes functions in response to events at any scale.
   - Users interact via a RESTful API:
   - Create Actions, Invoke Actions, Status Checks

2. NGINX: Receives API requests and routes them to the Controller.

3. Controller:
   - Task Scheduling, Resource Assignment, Sends tasks to Invokers via Kafka (distributed messaging).

4. Invoker:
   - Runs tasks in Docker containers.
   - Notifies Controller upon unloading containers.

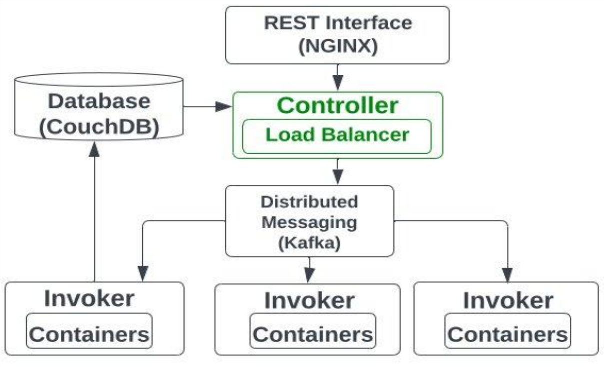


Fig: OpenWhisk architecture. Controller is where scheduling happens.

Scheduling of serverless functions & GPU Sharing

Scheduling of Serverless Functions:

1. Resource Assignment:
   - Determine the amount of resource (vCPUs, Memory) to assign to each of the serverless functions
2. Mapping to Invoker:
   - Choose an Invoker (computing node) based on the function's requirement
3. Meet the Service Level Objective (SLO)

GPU Sharing in Serverless Functions:

Sharing of single GPU resources by multiple processes

- Temporal sharing : Workloads executes in time-slices manner
- Spatial sharing : Workloads executes in a portion of GPU simultaneously
  - **NVIDIA MPS:** Shared resource with less isolation
  - **NVIDIA MIG:** Hardware level isolation in multiple isolated instances

NVIDIA: MPS & MIG

NVIDIA MPS:

- Allows multiple CUDA processes to share a single GPU without time slicing & execute concurrently
- Share the GPU's memory and compute resources but separate virtual memory address space
- Logical separation & low isolation

NVIDIA MIG:

- Hardware level separation & strong isolation
- Separate Streaming Multiprocessors (SM), memory, L2 cache, and bandwidth for each instance.
- Users can see and schedule jobs on virtual GPU instances as if they were physical GPUs.

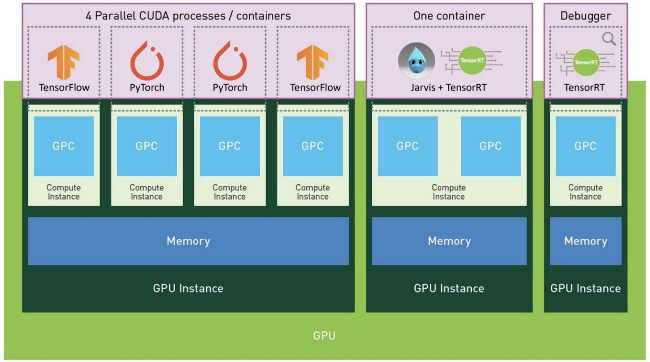


Fig: Multi-Instance GPU

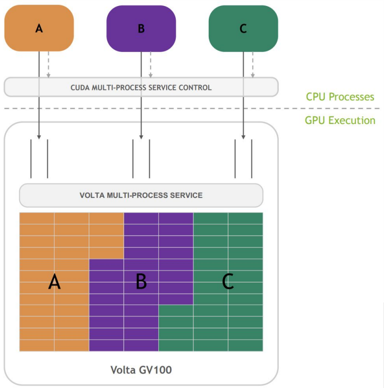


Fig: Multi-Process Service

ESG Scheduling Algorithm

Application-function-wise (AFW) job queues

- To group requests for the same serverless function of the same application together

- The AFW queues reside on the Controller
- Each of them gets populated as user requests arrive
- Or if its predecessor functions produce some triggering outputs

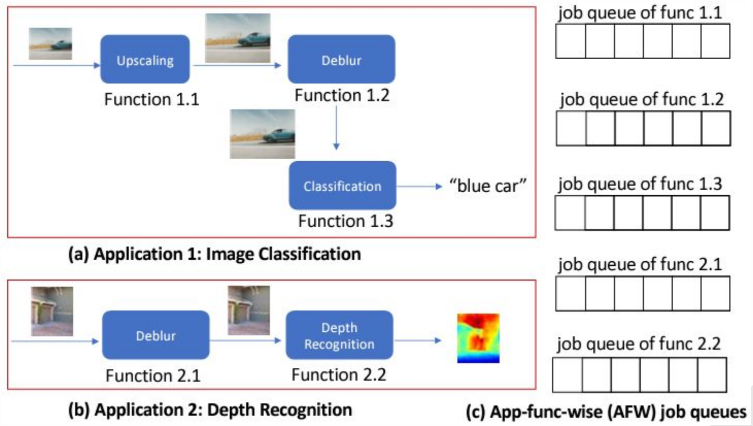


Fig: The app-func-wise (AFW) job queues of two example ML-based applications

ESG Scheduling Algorithm

Problem Formal Definition

Problem Setup

- Each job is a function invocation within an application.
- And we have a set of jobs during an inference call for an application
- The call has a tolerable latency upper limit
- We have set of workers nodes with certain CPU and GPU resources.

Objective

- Find schedule configurations (C) that specify:
  - Batch size, worker, and resource allocation.
- Minimize cost while meeting latency and resource constraints.

ESG Scheduling Algorithm

High-level workflow of the two core algorithms of ESG

ESG_1Q: Determines optimal configurations for jobs in a queue: Batch Size, no. of vCPUs & vGPUs.

ESG_Dispatch: Assigns tasks to Invokers (computing nodes).

- ESG_1Q configures jobs without checking current resource availability.
- ESG_1Q output multiple top configurations, forming a configuration priority queue
- ESG_Dispatch repeatedly dequeus the priority queue until suitable configuration is not found and assign the task to appropriate machine
- Handles dynamic changes in resource availability

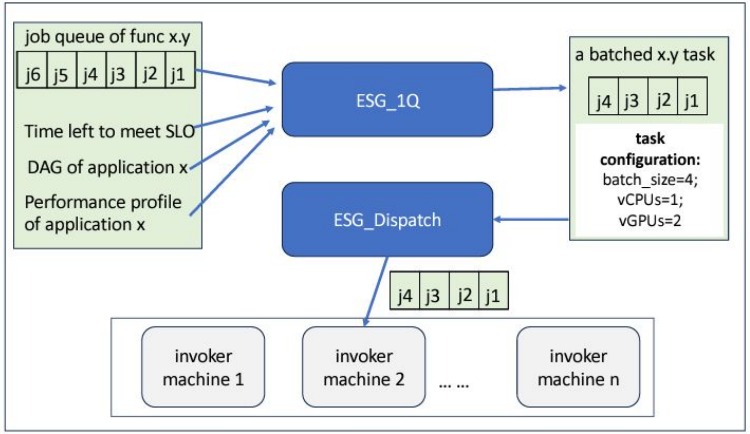


Fig: Workflow of the ESG scheduling algorithm on one job queue

ESG_1Q Algorithm

Path Selection: Each path represents a unique configuration (batch size, vCPUs, vGPUs) for each function.

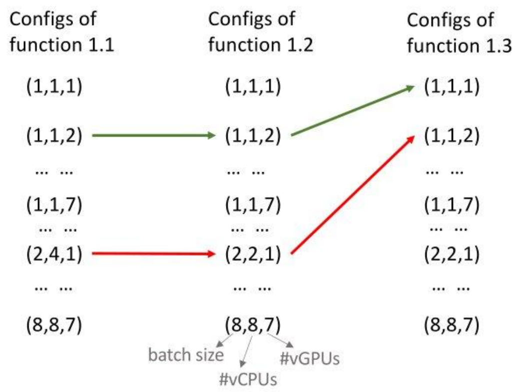
Goal: Find the path that meets SLO latency and minimizes resource costs.

Returns is not just a configuration good for the current function, but a sequence of configurations good for the whole application.

Uses A*-Search

A* is a best-first search algorithm which is both complete and optimal

- Dual-bladed pruning
- Dominator-based SLO distribution



(unit cost: 1 vCPU: 0.04¢/s; 1 vGPU: 0.8¢/s)

Paths	Time	Resource Cost per **Job**
Path 1 →	0.6+0.4+0.7=1.7s	0.98+0.66+0.59=2.23¢
Path 2 →	0.9+0.4+0.4=1.7s	0.43+0.18+0.66=1.27¢

(0.04*4+0.8)*0.9/**2**=0.43¢

Fig: Configuration space of an app and path costs

ESG_1Q Algorithm: Overview

1. **Initialize Path List:** Start with an empty list to store potential configuration paths.

2. **Iterate Over Functions:** For each function in the application:
   - **Sort Configurations:** Arrange available configurations by resource cost for efficient selection.
   - **Generate New Paths:** Extend existing paths by appending each sorted configuration.

3. Dual-Bladed Pruning:
- **Time-Based Pruning:** Eliminate paths that exceed the SLO latency threshold.
- **Cost-Based Pruning:** Discard paths that exceed the minimum cost among configurations.

4. Update Priority List:
- Track and update the best full paths found so far, prioritizing those with lower cost and within SLO constraints.

5. Return Best Paths:
- After processing all functions, return the best paths identified

ESG_1Q Algorithm

Dominator-based SLO Distribution for Scalability

Challenge: For the large workflows ESG algorithm's execution time can still be exceedingly long.

Goal: Divide SLOs among function groups to simplify scheduling while meeting latency requirements

1. Dominator Tree Creation:
    - Builds a tree structure of functions (DAG) as shown.
    - Helps identify independent function groups.
2. Stage Grouping:
    - Groups functions into manageable stages based on their dominator relationships.
3. SLO Assignment:
    - Assigns a specific SLO latency to each group, balancing end-to-end latency.

Then applies the ESG_1Q search algorithm to each individual group.

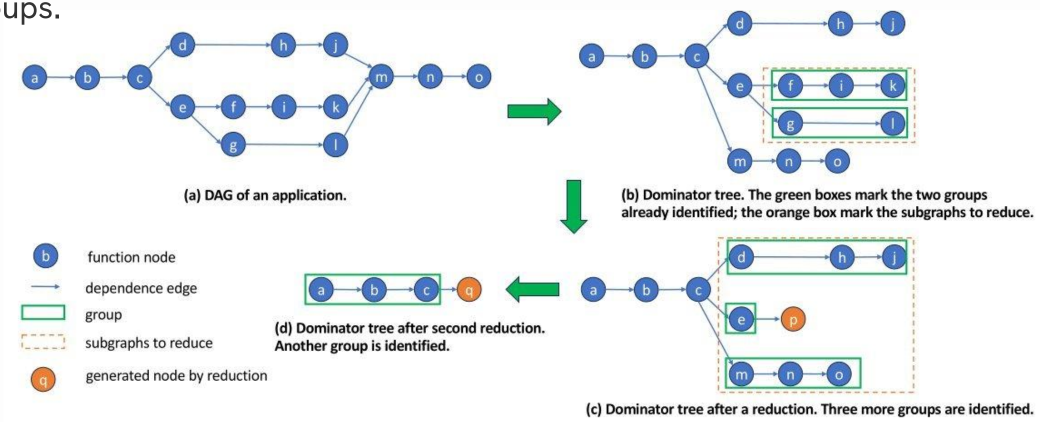


Fig: Illustration of dominator-based SLO distribution

ESG_Dispatch: Mapping to Worker Nodes

- ESG_1Q configures jobs without checking current resource availability.
- ESG_1Q output multiple top configurations, forming a configuration priority queue
- ESG_Dispatch repeatedly dequeus the priority queue until suitable configuration is not found
- ESG_Dispatch maps the current group of jobs to an Invoker node
- The algorithm chooses the *home-invoker* for the first function
- For other functions try to choose the invoker that runs its predecessor function in the workflow

**home-invoker:* the invoker where the future instances of the function will reside by default

Evaluation

Experimental Setup

Emulation Framework :

- **Purpose**: Simulates serverless workloads and scenarios on actual machines.
- **Workload Generator**: Generates tasks with various arrival rates to emulate real-world serverless demands.

Testbed : Consists of 16 nodes, each with:

- 16 vCPUs and 1 GPU (up to 7 vGPUs using NVIDIA MIG).

Workloads & Applications : Tested on four DNN applications with distinct workflows

- Image classification, Depth Recognition, Background elimination, Expanded image classification
- Job Arrival Rates: Light, normal, and heavy workloads modeled based on Azure traces.

SLO Settings : Strict, Moderate, Relaxed (Completes within 0.8*L, 1*L, 1.2*L)

Comparison with:

- **INFless** and **FaSTGShare**: the latest algorithms for sharable GPU-based serverless ML
- Best-first search algorithm in **Orion** and the Bayesian Optimization-based scheduling in **Aquatope**

End-to-End Performance

Performance and Cost Efficiency Comparison Across SLO Hit Rates

- High SLO Hit Rates: ESG achieves 46%-80% higher rates than BO/Orion and 36%-61% higher than INFless/Fast-GShare.
- Cost Efficiency: ESG offers better SLO rates with lower or similar costs, while INFless uses the most resources.

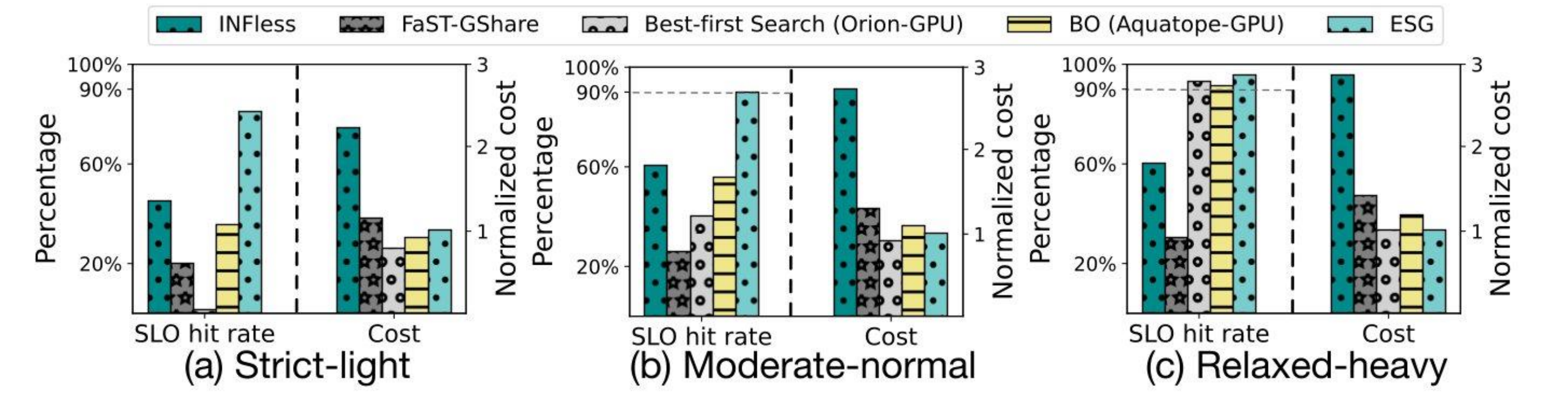


Fig: The average SLO hit rate and the cost(normalized to ESG cost) under different SLO and workload settings. The left y-axis is for the SLO hit rate and the higher is better. The righty-axis is for the cost and the lower is better.

End-to-End Performance

End-to-end latencies of each of the four applications

- ESG achieves latencies just below but close to the SLO latency
- Other methods like FaST-GShare leads to slower jobs and INFless leads higher resource usage

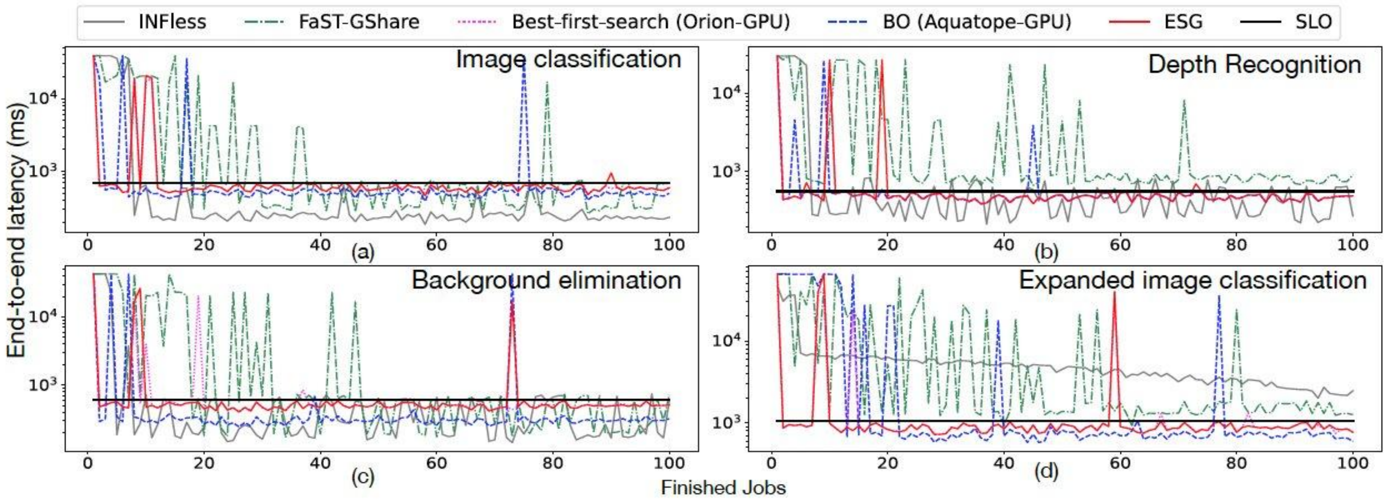


Fig: End-to-end latency for each application in relaxed-heavy setting

Overhead Analysis

Analysis of scheduling/searching overhead

Search Overhead: Less than 10 ms (Average shown by green triangle).

Effect of SLO Settings:

- Strict SLO: Lower overhead due to aggressive pruning.
- Relaxed SLO: Slightly higher overhead as more configurations meet the SLO, resulting in fewer pruning of paths.

Comparison to Brute-Force:

- ESG: Efficient with A* search and pruning.
- Brute-Force: Would take 7258 ms with 256 configurations per function.

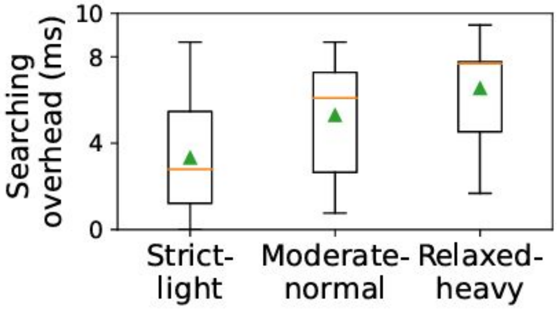


Fig: Scheduling overhead distribution of ESG (function group size is 3)

Detailed Analysis

Compared to INFless and FaST-GShare

- Distribute SLOs without accounting for inter-stage dependencies.
- Early stage delays (e.g., data transfer, cold starts) propagate to later stages, increasing overall latency.
- Higher latency and lower SLO compliance, shown by frequent SLO misses in figure.

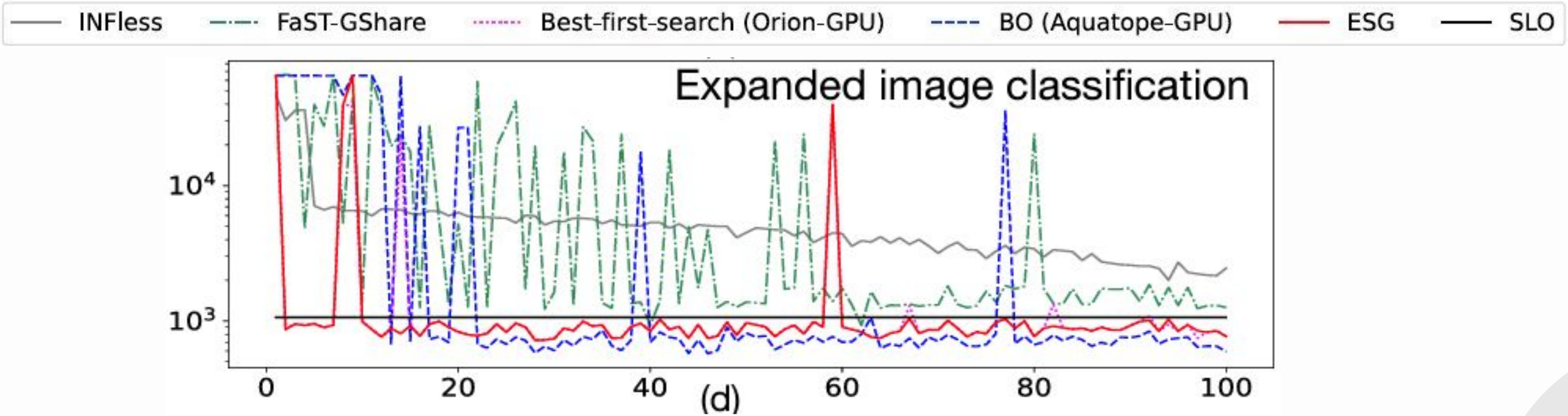


Fig: End-to-end latency for expanded image classification in relaxed-heavy setting

Detailed Analysis

Compared to Orion

- Orion is a search-based method, spend much time in searching
- Orion sets configurations for all functions while scheduling first function and doesn't adjust them for later stages
- ESG outperforms Orion as it finds better configurations much faster

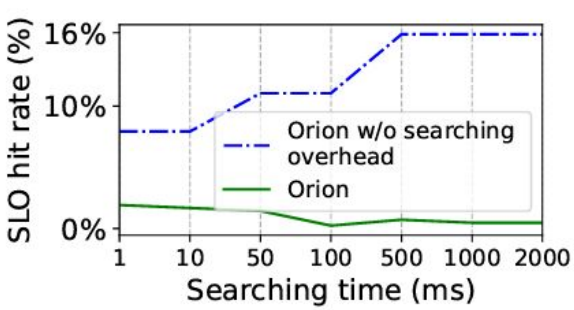


Fig: The effect of search time of Orion on the SLO hit rates (strict-light setting)

Compared to Aquatope

- Uses an offline-trained statistical model with minimal scheduling overhead.
- Relies on preset configurations that don't adapt to real-time changes in workloads.
- ESG dynamically adjusts configurations based on current conditions, leading to better performance and higher SLO compliance in changing environments.

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

Fig: Pre-planned scheduling miss rate

Contribution

- **Introduction of ESG:** First scheduling algorithm addressing inter-function relations, GPU sharing, batching, and runtime variations simultaneously.
- **Novel Optimizations:** Implements unique strategies to enhance efficiency and scalability in scheduling.
- **Resource Efficiency:** ESG minimizes resource use by dynamically adjusting configurations, balancing cost and performance for real-world applications.
- **Empirical Validation:** Demonstrates ESG's effectiveness through comparison with four state-of-the-art scheduling algorithms.

Limitations and Potential Improvement

- **NVIDIA MIG Availability :** This feature of GPU sharing (MIG) is only available in new and high-end GPUs (A100, H100, A30 ..)

- **Advanced Learning-Based Models:** Incorporate machine learning to predict optimal configurations based on past performance, enhancing scalability.

Thanks!

Do you have any questions?