FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service

(W6) Paper Reading

Nan Lin

Shanghai Jiao Tong University

2024-08-11

FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service

Zijun Li
Shanghai Jiao Tong University
Shanghai, China
lzjzx1122@sjtu.edu.cn

Yushi Liu
Shanghai Jiao Tong University
Shanghai, China
ziliuziliulys@sjtu.edu.cn

Linsong Guo
Shanghai Jiao Tong University
Shanghai, China
gls1196@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Jiagan Cheng
Shanghai Jiao Tong University
Shanghai, China
chengjiagan@sjtu.edu.cn

Wenli Zheng
Shanghai Jiao Tong University
Shanghai, China
zheng-wl@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

- ASPLos ‘22, February 28 - March 4, 2022
- Keywords: *Decentralization*

Outline

Review

Motivation

Architecture

Evaluation

Outline

Serverless Workflow

Serverless Workflow

Serverless functions are event-driven, and they need to be executed in a pre-defined order. Such a diagram with nodes connected by edges in a DAG form is known as the **serverless workflow**.

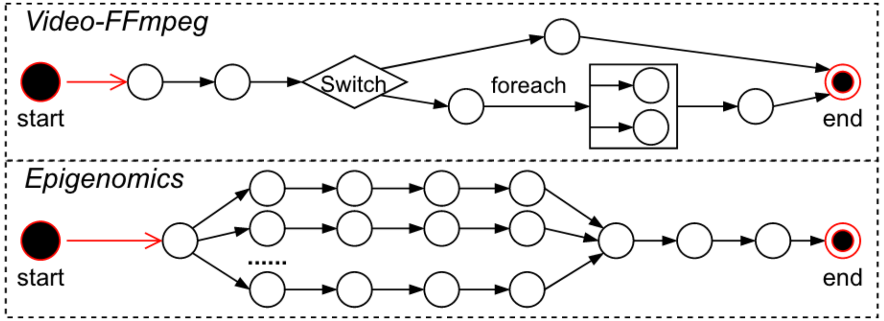


Figure 2: The example DAG-based workflows.

Serverless Workflow

Control-Plane and Data-Plane

- **Control-Plane**: User-defined execution order
- **Data-Plane**: Runtime data dependency

Usually identical and static. However, in serverless context, auto-scaling and warm containers may lead to multiple and different scales in the data-plane.

Outline

MasterSP and its Limitations

Master-side Workflow Schedule Pattern

Centralized Workflow: Central workflow engine in the master node determines whether a function task is triggered to run or not.

- Engine makes resource provision
- Task T_f triggered only if its predecessors are all completed:
  1. Assign T_f from the master engine
  2. Execute T_f invocation
  3. Return the exectuion state to the master engine

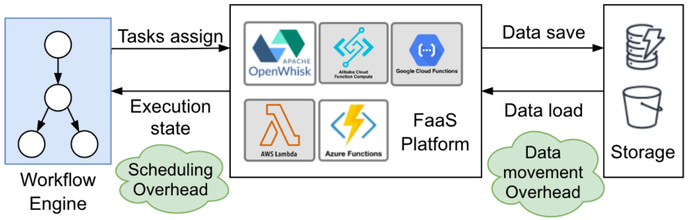


Figure 1: The overhead analysis for traditional workflow execution architecture in serverless context. WorkerSP

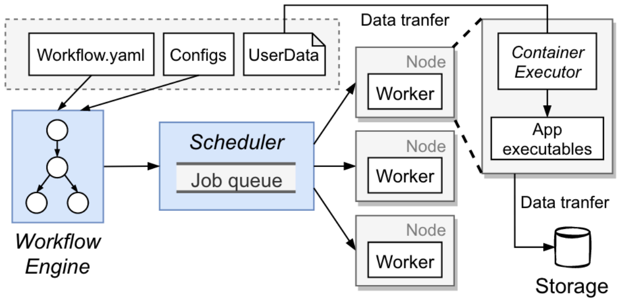


Figure 3: Implemented prototype of HyperFlow-serverless.

MasterSP and its Limitations

Problems:

- Large scheduling overhead: Transfer of function execution states
- Large data movement overhead: Additional database storage services for temp data storage and delivery

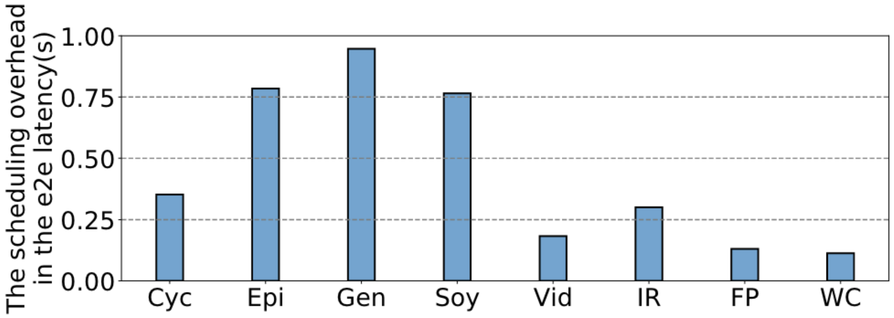


Figure 4: The scheduling overhead of executing the workflow benchmark (the scheduling overhead and the end-to-end latency depend on the critical path).

Data-Shipping Pattern

Data-Shipping Pattern

Each time a function task runs, the input data needs to be fetched from its predecessor functions, then read into memory for execution by the container executor. Such process is called a **data-shipping pattern**.

Problems:

- Function isolation brings more overhead of task-to-task data communcation
- Compulsory for user to use remote storage services
- Data locality is not utilized

Outline

WorkerSP's Structure Organization

The inverse of MasterSP: **Worker-side workflow schedule pattern (WorkerSP)**, *Decentralize*

Structure Organization of WorkerSP

- Master node scheduling $\xrightarrow[\text{Offload}]{}$ Per-worker engine assigned to perform local function triggering and invoking
- Master node only partition a workflow graph into sub-graph (See later)
- *Workflow* structure introduced with *State*, *FunctionInfo* and *InvocationID*.
  - *State*: Execution state of functions and their predecessors for invocation synchronization
  - *FunctionInfo*: Meta information for local functions
  - *InvocationID*: Unique state identification

WorkerSP's State Synchronization

(Reminder) Engine of each worker node maintains functions' and their predecessors' execution state in the **local sub-graph**.

Example: Invocation Synchronization

1. F_A is invoked
2. State pass to Node B and C
3. F_B and F_C update info
4. When F_A finished, `PredecessorsDone` of B and C + 1
5. When `PredecessorsDone` = `PredecessorsCount`, local engine of B and C will trigger

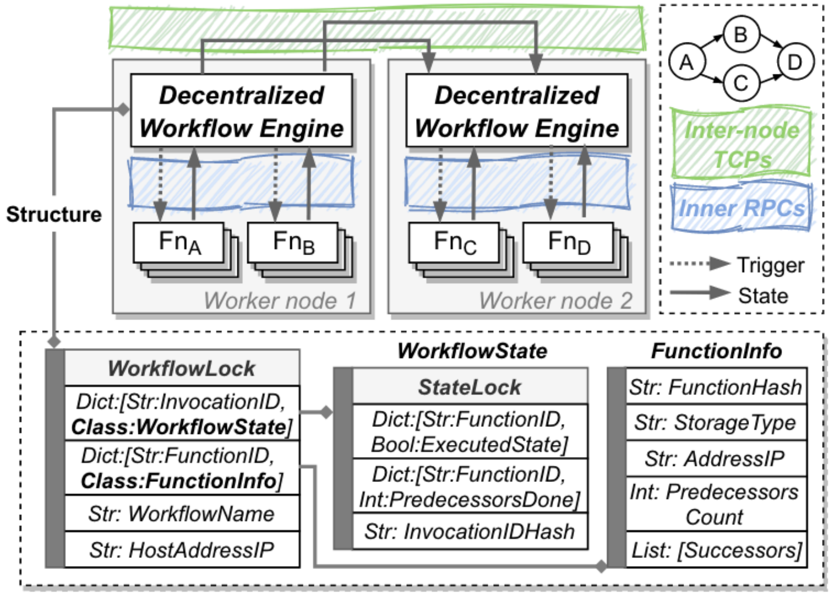


Figure 6: The data structure for workflow triggering and invocation management in the WorkerSP.

Overview of FaaSFlow

FaaSFlow: Workflow System

Three components:

1. **Workflow graph scheduler**
2. **Per-worker workflow engine**
3. Adaptive Storage Library **FaaStore**

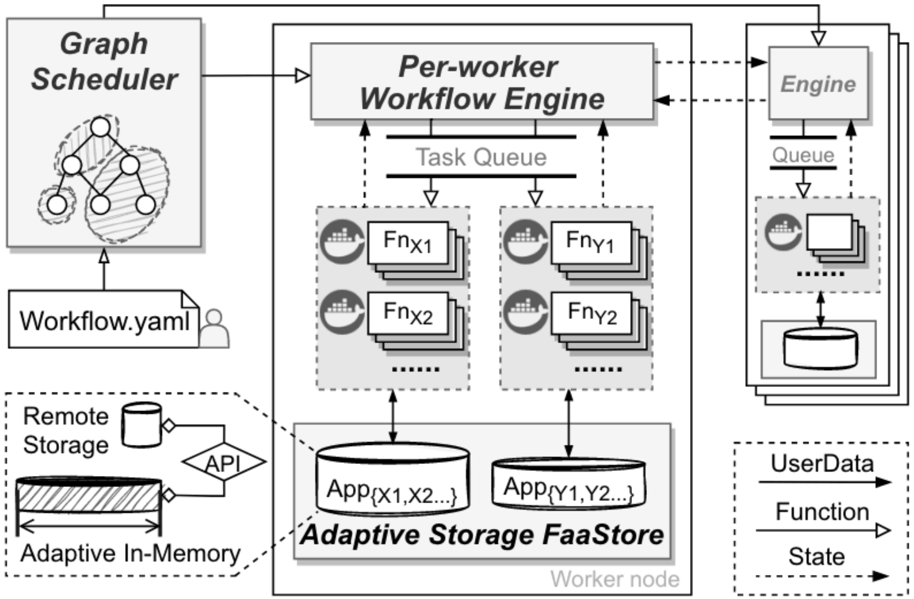


Figure 8: System design of FaaSFlow.

Component 1: Graph Scheduler

1. **DAG Parser** parse the hierarchy *Workflow Definition Language (WDL)* (which defines a serverless workflow)

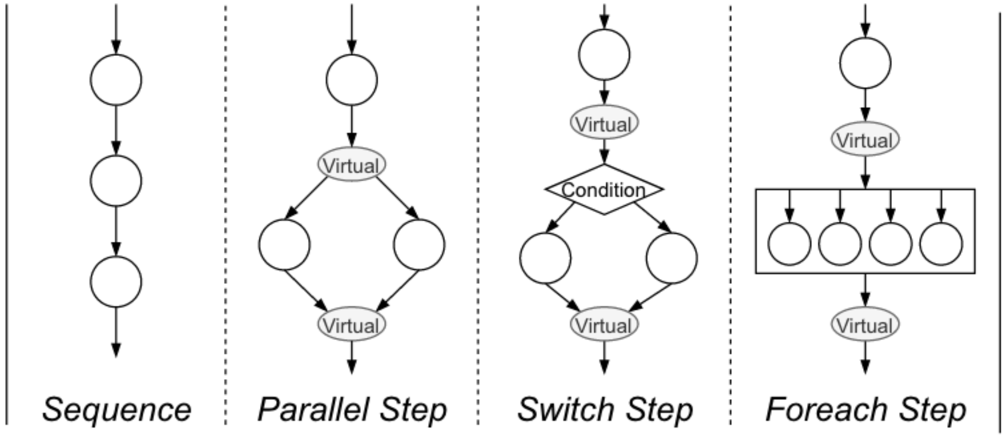


Figure 9: The supported logic flows in FaaSFlow and virtual nodes introduced in the parsing steps.

Component 1: Graph Scheduler

2. **Graph Partitionning**: Partitioning of DAG

To alleviate the gap between Control-plane and (dynamic data-plane):

- $\overline{\mathrm{Scale}(v_i)}$: Avg. number of scaled instances of a function node v_i during iteration
- $\overline{\mathrm{Map}(v_i)}$: Mapped instances in the data-plane (e.g. `Foreach`)

Partition iteration activated when significant performance degredation

Algorithm 1: Functions grouping and scheduling.

Data: $Cap[node]$: a list of the capacity of containers left to be created on each node. $Quota(G)$: the in-memory storage quota of graph G, discussed in Section 4.3.1.

Input: workflow graph G, functions $f_1, f_2, ..., f_n$

Each node as a group

1

$S \leftarrow \{\{f_1\}, \{f_2\}, ..., \{f_n\}\}$; $W \leftarrow$ RandomNodes(S);

$\{f\}.StorageType \leftarrow$ 'DB'; $mem\_consume \leftarrow 0$;

repeat

Locate functions with longest edge

2

  cpath = $(\{f\}, \{e\}) \leftarrow$ critical_path(G);

  descend_sort_by_weight({e}); $flag_{merge} \leftarrow$ False;

  for e in $\{e\}$ **do**

    $f_{start} \leftarrow e.start$, $f_{end} \leftarrow e.end$;

Merge

3

    $S_{start} \leftarrow S[S.\mathrm{find}(f_{start})]$, $S_{end} \leftarrow S[S.\mathrm{find}(f_{end})]$;

    If $S_{start} == S_{end}$ **then** continue; **end**

    $n_{start} \leftarrow \sum_{s_i}^{S_{start}} Scale(s_i)$, $Cap[W(S_{start})]$ -= n_{start};

Do not exceed Max capacity

4

    $n_{end} \leftarrow \sum_{s_j}^{S_{end}} Scale(s_j)$, $Cap[W(S_{end})]$ -= n_{end};

    If $n_{start}+n_{end} > max(Cap[node])$ **then** continue; **end**

Do not exceed memory constr.

5

    if $f_{start}.StorageType$ == 'DB' **then**

      If $mem\_consume + e.weight > Quota(G)$ **then**

        continue; **end**

      $mem\_consume+ = e.weight$;

      $f_{start}.StorageType \leftarrow$ 'MEM';

    end

Avoid memory contention functions

6

    If $(f_i, f_j) \subseteq S_{start} \cup S_{end}$ **and** $(f_i, f_j) \not\subseteq$ cont(G) **then**

      $S_{new} \leftarrow S_{start} \cup S_{end}$; **else** continue; **end**

    $W[S_{new}] \leftarrow$ binpack($limit$=$Cap[node]$>$n_{start}+n_{end}$);

    $Cap[W(S_{new})]$ += $n_{start} + n_{end}$;

    S.insert(S_{new}), S.delete(S_{start}, S_{end});

    $flag_{merge} \leftarrow$ True; break;

  end

until $flag_{merge} == False$;

return group $S = \{S_1, S_2, ..., S_k\}$, Worker $W[S_i] = node$

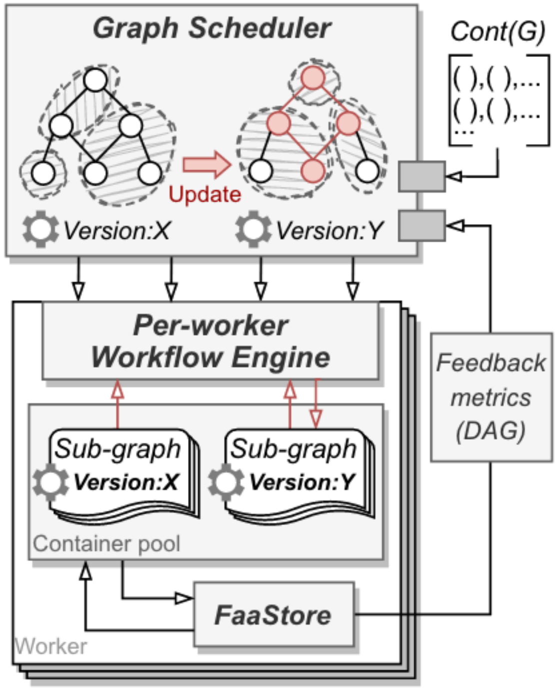
Component 2: Per-Worker Workflow Engine

Maintaining states for different functions.

Direct state communication via **inter-node TCP** or **inner RPC connections**.

Red-Black Deployment

Manage different versions of sub-graph versions in worker engines, only the up-to-date version is getting triggered.



(a) Red-Black Deployment

Component 3: FaaStore

In-memory storage enables data and files reside in local sub-graph; defUlt remote store save them by user configs.

In-Memory Quota

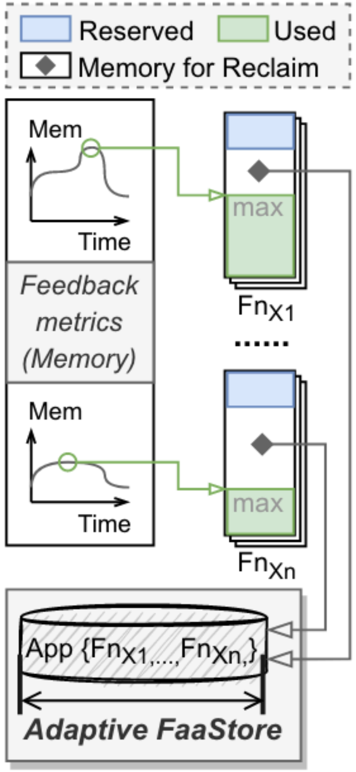
Well-organized quota for data movement

Due to *over-provisionning*,

$$O(v_i) = \max(\mathrm{Mem}(v_i) - S - \mu, 0)$$

and

$$\mathrm{Quota}(G(V, E)) = \sum_{v \in V} O(v)$$



(b) Memory Reclamation

Outline

Evaluation

FaaSFlow reduces the scheduling overhead from 712ms to 141.9ms for scientific workflows, and from 181.3ms to 51.4ms for real-world applications on average. All applications can achieve an average of 74.6% scheduling overhead optimization in FaaSFlow.

Basically did not mention memory allocation improvements ……