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

ABSTRACT

Serverless computing (Function-as-a-Service) provides fine-grain resource sharing by running functions (or Lambdas) in containers. Data-dependent functions are required to be invoked following a pre-defined logic, which is known as serverless workflows. However, our investigation shows that the traditional master-worker based workflow execution architecture performs poorly in serverless context. One significant overhead results from the master-side workflow schedule pattern, with which the functions are triggered in the master node and assigned to worker nodes for execution. Besides, the data movement between workers also reduces the throughput.

To this end, we present a worker-side workflow schedule pattern for serverless workflow execution. Following the design, we implement FaaSFlow to enable efficient workflow execution in the serverless context. Besides, we propose an adaptive storage library *FaaStore* that enables fast data transfer between functions on the same node without through the database. Experiment results show that FaaSFlow effectively mitigates the workflow scheduling overhead by 74.6% on average and data transmission overhead by 95% at most. When the network bandwidth fluctuates, FaaSFlow-FaaStore reduces the throughput degradation by 23.0%, and is able to multiply the utilization of network bandwidth by 1.5X-4X.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507717

KEYWORDS

FaaS, serverless workflows, graph partition, master-worker

ACM Reference Format:
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 (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507717

1 INTRODUCTION

Serverless computing (Function-as-a-Service, FaaS[1]) is widely accepted in the cloud-native era. Adopting serverless computing, the tenants submit functions directly without renting virtual machines of different specifications, and the vendors schedule the functions automatically and efficiently [13, 46, 49, 52, 60, 63]. While some serverless applications still use simple functions, emerging applications start to use fine-grained functions to provide complex functionality by connecting them into workflows [20, 61]. In this scenario, the serverless functions run in user pre-defined logic order, representing the workflow's control-plane. These serverless workflows usually contain parallel branches and data dependencies, and are logically defined through DAG (Direct Acyclic Graph) [18, 23, 24, 39, 57, 65].

Main cloud vendors provide serverless workflow products, such as AWS Step Funcions [2], Microsoft Durable Functions [5], Google Workflows [8], and Alibaba Serverless Workflows [1]. The open-sourced serverless systems OpenWhisk [9] and Fission [7] also support workflow for processing function sequence. These serverless workflow systems usually provide a centralized workflow engine on the master node to manage the workflow execution state and assign function tasks to the worker nodes. We refer to this scheduling pattern as *master-side workflow schedule pattern (denoted by MasterSP)*, as the central workflow engine in the master node determines whether a function task is triggered to run or not.

[1]We use both "serverless" and "FaaS" in the following paper.

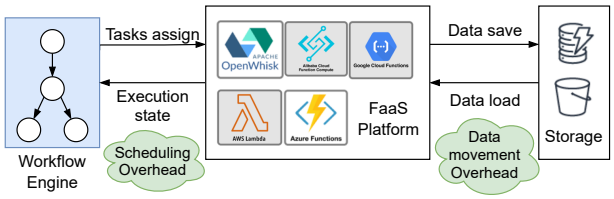


Figure 1: The overhead analysis for traditional workflow execution architecture in serverless context.

However, MasterSP does not match well with the event-driven feature of the serverless computing. Figure 1 shows the way of using MasterSP to manage a serverless workflow. As observed, MasterSP results in poor performance of serverless workflow due to the large *scheduling overhead*, and the large *data movement overhead*. On the one hand, the central workflow engine is responsible for dynamically managing and scheduling all functions. The function execution states are frequently transferred from the master node to the worker nodes, introducing heavy scheduling overhead. Since a function is short, the transfer happens frequently. On the other hand, the engine "randomly" distributes the triggered functions to worker nodes for load balance, and cloud vendors impose quotas on the input and output data size of functions to avoid serious consumption of network bandwidth [10–12, 31]. In production serverless platforms, users often rely on additional database storage services for temporary data storage and delivery, suffering significant data movement overhead [36, 37, 39, 55, 56].

In order to support serverless workflow execution more efficiently, we propose a *worker-side workflow schedule pattern (WorkerSP)* to alleviate the scheduling overhead. In WorkerSP, the master node offloads the task scheduling to each worker node, and is only responsible for workflow graph partition. On each worker node, the per-worker workflow engine independently judges whether it can trigger a local function based on the execution state of its predecessor functions received from other nodes. It is nontrivial to achieve such an efficient worker-side workflow scheduling, as a function worker may perform the auto-scaling and reuse warm containers, which results in that each function node in the control-plane may have multiple and different scales in the data-plane. In the case that the actual control-plane (user pre-defined) and data-plane (data dependency) of a serverless workflow are not necessarily the same, it is challenging when partitioning a large-scale workflow to multiple workers at two different planes unless otherwise specified. Furthermore, considering the premise that resources dynamically change in the cluster, a mechanism is also required to partition and schedule a workflow based on real-time resource availability.

Besides, we propose *FaaStore*, an adaptive storage library that alleviates the expensive data movement overhead of remote storage. With *FaaStore*, functions on the same worker node are allowed to communicate directly through the shared main memory instead of the remote storage. However, leveraging main memory to exchange data between functions bumps into the challenge of properly allocating additional memory space. Typically it serves as a subscription with the empirical quota without theoretical guidance. An unorganized and small quota cannot take full advantage of the efficiency of data locality. In contrast, a large quota will add memory pressure on the physical node, and even cause memory swap or serious OOM (Out-Of-Memory) error in the worst case.

Based on the principles of WorkerSP and *FaaStore*, we propose and implement an efficient serverless workflow system **FaaSFlow**. FaaSFlow supports defining complex serverless workflow using "foreach", "switch" etc., by proposing a *workflow parser*. With FaaSFlow, a *workflow graph scheduler* runs on the master node. The graph scheduler resolves user-uploaded workflows, partitions the workflows into subgraphs based on the available resources on each worker node and the amount of data transferred between adjacent functions. On each worker node, FaaSFlow runs a *per-worker workflow engine* that manages the function state and triggers local function tasks, and an integrated *FaaStore* dynamically allocating the over-provisioned memory in the container at runtime. *FaaStore* uses appropriate data store (either well-allocated main memory from containers, or the remote store) to support the communications based on the locations and dependencies of the functions.

The main contributions of this paper are as follows.

(1) **The WorkerSP design and the adaptive library *FaaStore* in the master-worker computing architecture.** We explore the weakness of current serverless workflow framework based on MasterSP. WorkerSP can efficiently reduce the scheduling overhead, and *FaaStore* can alleviates the expensive data movement overhead.
(2) **A WorkerSP-based serverless workflow system FaaSFlow.** Based on WorkerSP, FaaSFlow re-organizes the workflow data structure to implement several logic flows, such as sequence, parallel, switch, and foreach steps.
(3) **A WorkerSP-based workflow graph partition strategy by function grouping.** When grouping functions into sub-DAGs, we consider both the available memory in each worker and function instances scaling. Function groups can be scheduled to execute on the appropriate worker.

Our experimental results indicate that the WorkerSP-based FaaSFlow reduces the scheduling overhead by 74.6% on average, and data movement by 95% at most. It also reduces the throughput degradation by 23.0%, and is able to multiply the utilization of network bandwidth by 1.5X-4X.

2 SERVERLESS WORKFLOW BACKGROUND

In this section, we introduce state-of-the-art systems for serverless workflow, and analyze their efficiency.

2.1 Serverless Workflows and Benchmarks

The event-driven feature of serverless functions makes them popular for stateless applications, such as web applications, IoT (Internet of Things), media, and file processing. For instance, emerging cloud applications are consist of hundreds of functions (e.g., a real-world social network service is implemented by connecting around 170 functions [19]). These functions need to be executed in a pre-defined order by the workflow to implement the complex business logic.

For a workflow built from serverless functions, the workflow can be mapped into a diagram with nodes connected by edges in a DAG form. It is also known as the serverless workflow [18, 22, 23, 40]. Most cloud vendors only support sequential workflow, which is

Table 1: The Benchmarks Used in the Experiment

	Benchmarks
Scientific workflows from Pegasus [3, 38]	Cycles, Epigenomics, Genome, SoyKB
Real-world applications with open-sourced	Video-FFmpeg (Alibaba Function Compute), Illgal Recognizer (Google Cloud Functions), File Processing (AWS Lambda), Word Count

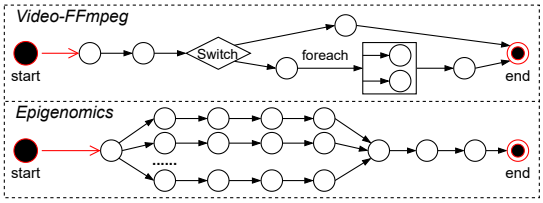


Figure 2: The example DAG-based workflows.

a much simpler execution model. In such a function sequence, functions are invoked following a serial pipeline, and receive the output from its predecessor as input.

Since function sequence is much simpler than DAG-based workflows, this work focus on complex DAG-based workflows. To investigate the execution efficiency of serverless workflows, we select four popular scientific workflows and implement four representative serverless workflow applications from AWS, Google, and Alibaba as our benchmarks. As an example, Figure 2 shows the DAGs of two benchmarks. They are not simple function sequences.

- **Cycles (Cyc), Epigenomics (Epi), Genome (Gen), SoyKB (Soy),** are the scientific workflow execution instances generated from Pegasus workflow executions [3, 27, 28, 38], which is a workflow management system for executing DAG-based workflows. All these four scientific workflows are configured with 50 functions nodes.
- **Video-FFmpeg (Vid)** is an FFmpeg-based audio and video processing application. Function calls FFmpeg to parallelly transcode the video uploaded and return it. Alibaba Function Compute provides such a use case [17].
- **Illegal Recognizer (IR)** is a composite application that extracts text from the uploaded image, then translates the text using the Google Cloud Translation API, and finally detects and blurs offensive in the image. Its source code comes from Google Cloud Functions [15, 16].
- **File Processing (FP)** is a real-time file processing application, which delivers notes from the database and then converts to HTML and detects sentiment in parallel. Its source code comes from AWS Lambda use cases [14].
- **Word Count (WC)** is the most classic and common application that implemented with reference to Zhang *et al.* [64].

2.2 Master-Side Workflow Schedule Pattern

Traditional workflow model [18, 20, 21, 30, 43] follows a standard master-worker pattern. In this case, the master node is needed in a workflow engine, which maintains the functions and workflows states. Master node collects the execution states of functions from

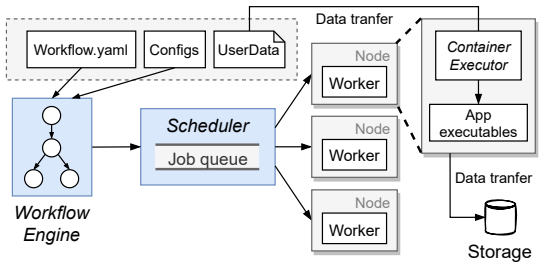


Figure 3: Implemented prototype of HyperFlow-serverless.

the worker nodes and determines whether functions in the workflow meet their trigger conditions. Once predecessors of function f are all completed, task T_f will be triggered and assigned to a worker node for invocation, and returned with the execution state. This workflow execution pattern is referred to as the **master-side workflow schedule pattern (MasterSP)**.

State-of-the-art workflow engine for functions with complex dependencies is HyperFlow [21]. However, it is only applicable for the serverful-based workflow architecture. Based on the idea from Malawski *et al.* [43], we port it to the serverless context, and implement the HyperFlow prototype system (denoted by *HyperFlow-serverless*). Figure 3 shows the way that HyperFlow-serverless manages the serverless workflows, which also uses the MasterSP to trigger the execution of functions. In HyperFlow-serverless, a central workflow engine is deployed in the cluster and invokes functions by assigning tasks to different worker nodes. The HyperFlow-serverless engine receives the workflow objects through *workflow.yaml*, and makes resource provision for containers in the worker node. The user data is transferred to the container executor after task assignment, and then a common storage database will save the execution result.

Observed from Figure 3, the master engine with MasterSP still needs to collect the states and check execution trigger conditions, even if a function and its predecessors are on the same worker node. It results in a large number of useless network interactions and long response latencies between functions.

2.3 Overhead of MasterSP

We use the benchmarks in Section 2.1 to investigate the scheduling overhead in HyperFlow-serverless. We use one node to generate workflow invocations and deploy the central workflow engine, and the other 7 nodes as the workers to perform the computing. The hardware and software setups are described in Section 5.1.

In this experiment, all required input data for functions is prepared and packed in the container image. We do not use open-loop control when measuring end-to-end invocation latency because functions may reuse warm containers and experience cold startup when fail to reuse one. Consequently, the queueing latency in an open-loop control will be inaccurately measured, resulting in a cascaded effect. Instead, we use one closed-loop client where it only tries to send another invocation after the execution state of the previous one has already been received. The measuring starts at the start of each loop, and ends when receiving the execution state.

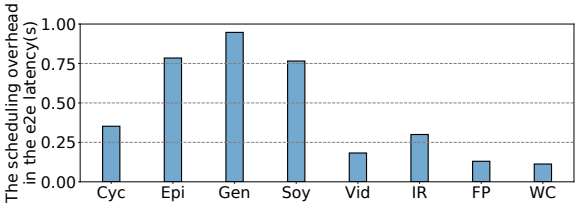


Figure 4: The scheduling overhead of executing the workflow benchmark (the scheduling overhead and the end-to-end latency depend on the critical path).

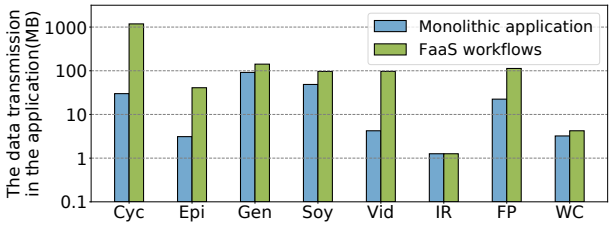


Figure 5: The data movement required for applications when deployed in monolithic and FaaS architecture.

By such means, the end-to-end invocation latency avoids the potential cold startup impact on the queuing latency. It includes only the scheduling overhead from the workflow engine and the execution time of the function tasks in the critical path of the DAG. We calculate the scheduling overhead of invoking a workflow by deducting the execution time of these critical function nodes from the end-to-end latency. For each workflow benchmark, we independently make the closed-loop control with one client thread in HyperFlow-serverless and shows the result in Figure 4.

Observed from the figure, scientific workflow and the real-world application benchmarks introduce an average scheduling overhead of 712ms and 181.3ms, respectively. The overhead is long when a single function invocation is commonly short in FaaS. Scientific workflows suffer from larger scheduling overhead, because they have more function nodes. For latency-sensitive applications with millisecond-level latency targets, such as the current internet services, the scheduling overhead results in a poor user experience.

Each function invocation in MasterSP contains three stages: 1) assign function tasks from the master engine, 2) execute the function invocation, and 3) return the execution state to the master engine. The large scheduling overhead of MasterSP is significantly introduced in the first and last stages. It motivates us to design a new **worker-side workflow schedule pattern (WorkerSP)**, where functions can avoid the first stage of task assigning. In WorkerSP, functions on the same node use a decentralized engine to maintain the execution state, and directly perform cross-node state transfers without a master maintainer. In this case, the system can reduce network hops more efficiently than using a large number of concurrent TCP connections between the master workflow engine and the worker executors.

2.4 Data-Shipping Pattern

In serverless workflow, applications are built from fine-grained functions and executed by pre-defined logic steps. 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. Currently, most serverless systems use this data-shipping pattern [34].

In this subsection, we investigate the effectiveness of the data shipping pattern, by comparing Hyperflow-serverless with the scenario that all the functions run in the same server in a monolithic way. In this case, all functions running in the same container use direct inter-calls, and the required data is passed without a database.

Table 2: The Hard Quota for each Request Size (including all request data) of Popular Serverless Platforms

Serverless Platform	Hard Quota (per request)
AWS Lambda	6MB(synchronous), 256K(asynchronous)
Google Cloud Functions	10MB for data sending to functions
Microsoft Azure Functions	1MB with single stream
Alibaba Function Compute	6MB(synchronous), 128K(asynchronous)
Apache OpenWhisk	1MB for each entity

By replaying the experiment above, we show the amount of data movement for each workflow invocation in Figure 5.

As shown in the figure, it is indubitable that function isolation brings more overhead of task-to-task data communication than directly building a monolithic application. Particularly, the required data movement of *Vid* and *Cyc* soars from 4.23MB and 23.95MB to 96.82MB and 1182.3MB, respectively, after deploying the application into a serverless workflow. *Vid* and *Cyc* require 22.86X and 39.46X higher network resources when executing them under the serverless (FaaS) architecture.

Without considering data locality, even though two adjacent functions are in the same node, there is a massive overhead of making data transmission through an unnecessary database. Besides, due to limitations under the serverless architecture, the direct data transfer interface between functions does not provide enough data transmission quota. We list the quota from several cloud vendors that offer serverless executions, as shown in Table 2. Direct file or data transmission carring more than 6MB of data (10MB in Google Cloud Functions) will be throttled by could vendors. Furthermore, the quota for data transmission will be smaller in the workflow scenario [10–12], since the unrestricted data size will make bandwidth resources in the cluster violently consumed, thus leading to performance bottlenecks. It makes it compulsory for users to use remote storage services as an intermediate bridge when dealing with large files and large objects between function tasks.

It is necessary to enhance the data locality and reduce the data transmission through network for serverless workflows.

3 RATIONALE OF WORKERSP/FAASTORE

Worker-side workflow schedule pattern (WorkerSP) resolves the function triggering and state synchronization in each worker. *FaaStore* manages the hybrid storage for data fetching and saving in each worker.

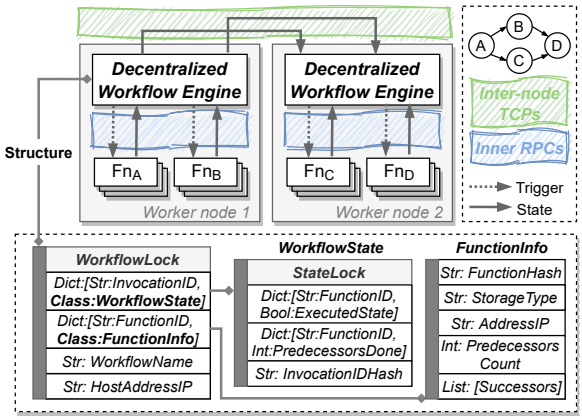


Figure 6: The data structure for workflow triggering and invocation management in the WorkerSP.

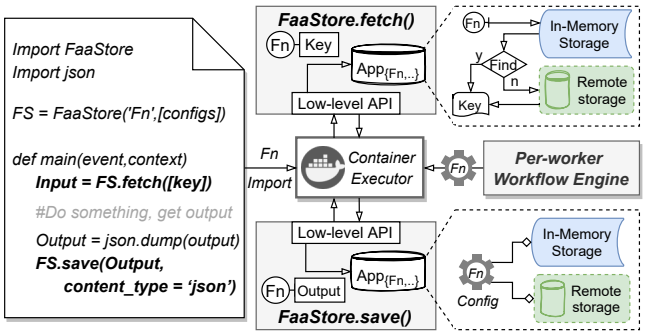


Figure 7: The co-design method of adaptive storage *FaaStore*, which cooperates with the local workflow engine and provides low-level APIs for storage interface.

3.1 Management and Synchronization

In WorkerSP, the graph scheduler in the master offloads the tasking assignment and state management. It only serves to partition a workflow graph into sub-graphs for worker nodes, while ensuring adequate resource allocation in workers for executing these sub-graphs. Per-worker engine will be assigned with sub-graphs to perform the local function triggering and invoking. Because that sub-graphs may be distributed across multiple workers, it is significant to re-organize the data structure for managing workflows and synchronizing invocations.

Structure organization in workers. The functions state of the workflow are decentralized and maintained in different worker nodes, and function tasks are triggered and invoked in the local worker node. To enable each worker to trigger functions in a local sub-graph independently, a *Workflow* structure is introduced with *State* and *FunctionInfo*, as the maintainer of this sub-graph. Each workflow invocation assigns an *InvocationID* to provide state identification from different invocations. In each worker *Workflow* structure, *State* preserves the execution state of functions and their predecessors for invocation synchronization and local triggering. *FunctionInfo* maintains the meta information for local functions in a specific workflow.

State synchronization between workers. Each worker handles synchronization of functions in its local engine by *State* and *FunctionInfo* structure, as shown in Figure 6. When all predecessors of a function are marked executed, the local engine triggers a local invocation, and synchronizes the updated *State* in its local sub-graph. If a function has cross-worker dependencies, the local engine will pass the executed state to the remote worker engine through TCP connections. The protocol for TCP communications is only responsible for updating function states, excluding assigning function tasks between the master and workers. The establishment of connections refers to the *FunctionInfo*, in terms of traversing the successor functions list for their IP addresses of scheduling location.

Take Figure 6 as an example to illustrate the WorkerSP for an invocation synchronization. The engine of each worker node maintains functions' and their predecessors' execution state in the local sub-graph. Once the function Fn_A is invoked, the engine updates the execution state into the local *Workflow*. Then, it will inspect Fn_A's successor functions Fn_B, Fn_C, and pass the execution states by inter-node TCP (e.g., distributed across different workers) or inner RPC connections (e.g., located on the same worker). Finally, function Fn_B, Fn_C will get updated of their predecessors' state in each worker engine, after which the number of executed predecessors is calculated. If the *PredecessorsDone* count of a function reaches its target *PredecessorsCount*, the local engine will trigger and invoke it locally.

3.2 Adaptive Storage with FaaStore

Separating the storage service from function computing services for business isolation makes application tenants pay extra attention. Code developers are required to register the remote storage services for large data movement, since RPC payloads cannot provide such a huge quota. To this end, WorkerSP enables the potential for hybrid storage using a co-design method of workflow engine and storage interface, which benefits from the graph dependencies maintained in each worker engine.

Dependence-driven hybrid storage. The Adaptive Storage Library *FaaStore* provides hybrid storage services and automatically uses the appropriate storage for functions. The in-memory storage enables data and files to reside in a local sub-graph; otherwise, a default remote store will save them by user configurations. Catering to the FaaS pattern, the in-memory storage should be designed with an adaptive quota for data cache, and should not result in additional memory stress in FaaS system. Through *FaaStore*, each worker node can independently localize and manage the workflow data movement, therefore reducing the unnecessary communication overhead.

Easy-to-use low-level API. We show the co-design mechanism of *FaaStore* in Figure 7. *FaaStore* replaces the standard remote database storage API in the user interface, when users have declared the *keys* in the workflow definition file *workflow.yaml*. In terms of serverless system architecture, the users' code is *read-only*, and the input and output information of the function invocation will not be queued alongwith the triggering of the function. Instead, the users'

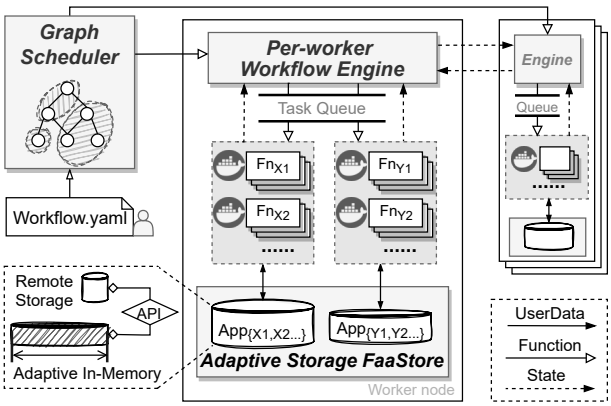


Figure 8: System design of FaaSFlow.

code and data should be imported into memory by the container executor. After which, *FaaStore* will inspect whether successors of this function locate on the same node, and accordingly select the appropriate data storage.

4 THE FAASFLOW SERVERLESS SYSTEM

Following the WorkerSP, we propose and implement an efficient serverless workflow system **FaaSFlow**. It comprises three components: (1) a *workflow graph scheduler* on the master node resolving user-uploaded workflows, partitions the workflows into sub-graphs based on the available resources on each worker node and the amount of data transferred between each function pairs; (2) a *per-worker workflow engine* on each worker node that manages the function state and triggers local function tasks; (3) an Adaptive Storage Library called *FaaStore* that dynamically allocates the over-provisioned memory resources in the container at runtime, and uses appropriate data store (either well-allocated memory from containers, or the remote database service) to support the communications based on the locations and dependencies of functions.

4.1 Graph Scheduler

The Graph Scheduler receives a *Workflow.yaml* configuration to define execution logic, and then parse it by the *DAG Parser* on the master node. By analyzing each worker node's resource usage, the scheduler can make appropriate graph partition decisions and deployment strategies.

4.1.1 DAG Parser for Workflows. The Workflow Definition Language (WDL) defines a serverless workflow, and inner data transmission by specified attributes *input* and *output* between steps. The DAG Parser is implemented in the Graph Scheduler to prevent violated WDL definition and parse the hierarchy WDL into a DAG object. FaaSFlow currently provides the following basic logic steps to describe and define an application logic:

- **Task (Function).** The task step defines the single function invocation. When a task step is invoked, the corresponding function is triggered. DAG Parser converts a single task step into a node in the DAG.

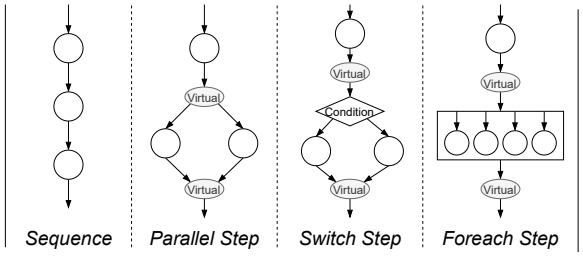


Figure 9: The supported logic flows in FaaSFlow and virtual nodes introduced in the parsing steps.

- **Sequence.** The sequence step contains multiple serial task steps in a flow. When a task step is executed successfully, the successor task step starts. DAG Parser takes serial steps as nodes and calculates the 99%-ile latency of data transmission between adjacent nodes as edge weight in the DAG.
- **Parallel.** The parallel step executes multiple child tasks in parallel, which is defined by attribute *branches* in the workflow. DAG Parser will perform the conversion for each branch in a parallel step.
- **Switch.** The switch step allows invocations of different task steps according to the conditional expression in a flow. Since the workflow still maintains function containers for all the branches in the switch step, the DAG Parser follows the same logic of a parallel step.
- **Foreach.** The foreach step traverses elements in the input and executes the task steps for each element in parallel. DAG Parser equally considers all parallel instances in the foreach step as one node, and denotes the slowest 99%-ile latency among all branches as the weight of the output edge.

Especially, we introduce a virtual start node and a virtual end node for parallel, switch, and foreach steps. The virtual node does not participate in any computation or states recording, and it only ensures atomic consistency of steps when partitioning sub-graphs.

4.1.2 Graph Partitioning in FaaS Context. A workflow graph has the control-plane (user-defined execution order) and the data-plane (runtime data dependency), which are normally identical and static. However, in the serverless context, they are not necessarily equal for the following significant reasons, and the graph partition should take the dynamic data-plane into considerations.

- **Auto-scaling and container reusing.** FaaS enables function nodes to perform auto-scaling and warm containers resuing in the workflow. Each node in the control-plane may have multiple and different scales in the data-plane. FaaSFlow introduces $\overline{Scale(v_i)}$ for each function node, which represents the average number of scaled instances of a function node v_i during partition iteration. This metric is updated based on the runtime feedback from the last iteration, and used for resource allocation in graph partition (Algorithm 1).
- **Multiple executors in foreach step.** The foreach step in the control-plane acts as an atomic operation, and may create multiple mapped instances in the data-plane. FaaSFlow introduces $\overline{Map(v_i)}$ for the foreach step, which represents the

average executors map of a function node v_i during partition iteration. It is also collected as a feedback metric at runtime ($\overline{Map(v_i)}$ equals 1 in other steps like parallel and switch). It is used to measure the in-memory quota in Section 4.3.1.

FaaSFlow leverages $\overline{Scale(v_i)}$ and $\overline{Map(v_i)}$ to resolve the above gaps between user-defined control-flow and the actual data-flow during each partition iteration. Each partition iteration is activated when the workflow experiences significant performance degradation or QoS violation. However, $\overline{Scale(v_i)}$ and $\overline{Map(v_i)}$ are unavailable when the workflow is invoked for the first time. To this end, FaaSFlow performs hash-based partition like other systems in the first partition iteration. *FaaStore* collects runtime metrics in each iteration, and updates the value/weight of the workflow DAG. Then, the graph scheduler will partition sub-graphs for worker engines. Section 4.2.2 gives a detailed discussion on the management of the per-worker workflow engine between different sub-graph iterations.

4.1.3 Function Grouping and Graph Scheduling. Considering that the partitioning of the DAG is an NP-hard problem [18, 45], here we do not need to search for the optimal solution. Instead, our purpose for functions partition is to search for the sub-optimal solution that meets the requirements with minimal response time. Therefore, we chose the greedy grouping strategy based on the critical path in FaaSFlow, as shown in Algorithm 1.

The key steps of the grouping and scheduling algorithm in the Graph Scheduler are illustrated as follows. Each function node is initialized as a separate group during the initialization phase, and the worker is randomly assigned (Line 1-2). Firstly, the algorithm starts with topo sort and iterates. At the start of each iteration, it will use the greedy method to locate two functions with the longest edge in the critical path of the DAG graph and determine whether two functions can be merged into the same group (Line 3-8). If these two functions are assigned in a different group, they will be merged (Line 9). When merging groups, additional factors need to be considered. Firstly, the algorithm needs to ensure that the merged functions group does not exceed the maximum capacity of the worker nodes (Line 10-12). Otherwise, the merged group cannot be deployed on any node. Secondly, the total amount of data localized within the group cannot violate the memory constraints (Line 13-18). Meanwhile, any function pair with resource contention $cont(G) = \{(f_i, f_j)\}$ cannot exist in the merged groups (Line 19-20). Finally, the scheduling algorithm will adopt the bin-pack strategy to select appropriate work nodes for each function group according to their node capacity (Line 21-23). Following the above logic, the algorithm iterates until convergence, indicating that the function groups are no longer updated.

Because functions may have different resource sensitivities, the load balancer should avoid co-locating heavy function workloads that are sensitive to the same resource. To this end, the Graph Scheduler should also take the resource contention between function nodes into account when grouping and scheduling. It should be noticed that we do not introduce additional strategies for resolving function interference for that there is already a lot of work solving this problem [25–27, 33, 44, 62]. By providing integration in the load balancer, FaaSFlow can also cooperate with these orthogonal works above by declaring conflict function pairs $cont(G) = \{(f_i, f_j)\}$.

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

1 $S \leftarrow \{\{f_1\}, \{f_2\}, ..., \{f_n\}\}$; $W \leftarrow$ RandomNodes(S);
2 $\{f\}.StorageType \leftarrow$ 'DB'; $mem\_consume \leftarrow 0$;
3 **repeat**
4 cpath = ($\{f\}$, $\{e\}$) $\leftarrow$ critical_path(G);
5 descend_sort_by_weight({e}); $flag_{merge} \leftarrow$ False;
6 **for** *e in {e}* **do**
7 $f_{start} \leftarrow e.start$, $f_{end} \leftarrow e.end$;
8 $S_{start} \leftarrow S[S.\text{find}(f_{start})]$, $S_{end} \leftarrow S[S.\text{find}(f_{end})]$;
9 **If** $S_{start} == S_{end}$ **then** continue; **end**
10 $n_{start} \leftarrow \sum_{s_i}^{S_{start}} Scale(s_i), Cap[W(S_{start})]$ -= n_{start};
11 $n_{end} \leftarrow \sum_{s_j}^{S_{end}} Scale(s_j), Cap[W(S_{end})]$ -= n_{end};
12 **If** $n_{start}+n_{end} > max(Cap[node])$ **then** continue; **end**
13 **if** $f_{start}.StorageType ==$ *'DB'* **then**
14 **If** $mem\_consume + e.weight > Quota(G)$ **then**
15 continue; **end**
16 $mem\_consume+ = e.weight$;
17 $f_{start}.StorageType \leftarrow$ 'MEM';
18 **end**
19 **If** $(f_i, f_j) \subseteq S_{start} \cup S_{end}$ **and** $(f_i, f_j) \notin$ cont(G) **then**
20 $S_{new} \leftarrow S_{start} \cup S_{end}$; **else** continue; **end**
21 $W[S_{new}] \leftarrow$ binpack($limit$=$Cap[node]$>$n_{start}+n_{end}$);
22 $Cap[W(S_{new})]$ += $n_{start} + n_{end}$;
23 S.insert(S_{new}), S.delete(S_{start}, S_{end});
24 $flag_{merge} \leftarrow$ True; break;
25 **end**
26 **until** $flag_{merge} == False$;
27 **return** *group S = {S_1, S_2, ..., S_k}, Worker W[S_i] = node*

4.2 Per-Worker Workflow Engine

In WorkerSP, only the state of a function is communicated across different worker nodes. In this case, each worker engine is assigned with sub-graphs and can trigger functions locally, rather than receiving functions tasks from the master node.

4.2.1 Decentralized Engines. For traditional jobs, decentralized schedulers like Omega [51] and Sparrow [48] only make resource scheduling to find a more appropriate strategy, and they do not consider the data dependencies between different jobs or tasks in a decentralized way. However, serverless workflows are more concerned about maintaining the execution states for different functions, and the MasterSP in decentralized schedulers above may still introduce heavy traffic overhead. Therefore, the key of decentralized workflow engines is how to enable efficient execution state synchronization and communication for serverless workflows.

Using the WorkerSP mechanism of scheduling management and synchronization described in Section 3, the Per-worker Workflow Engine enables direct states communication between different workers. At the start of each partition iteration, all worker engines will be assigned with updated sub-graphs, and then perform the WorkerSP for serverless workflow invocations. In terms of proper memory

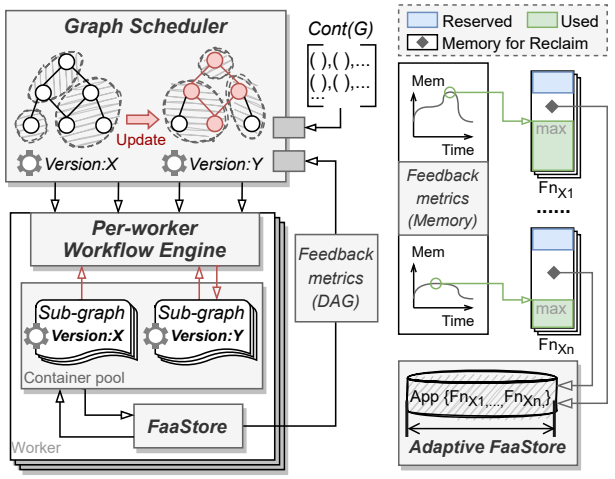


Figure 10: Feedback-based runtime scheduling mechanism.

consumption and utilization in WorkerSP, the per-worker engine should release the *State* object at the end of each invocation, and remove the local *Workflow* structure once all the function node instances in this workflow are recycled.

4.2.2 Red-Black Deployment and Management. When the Graph Scheduler refers to conflict function pairs and feedback metrics in each partition iteration, the sub-graphs are updated for each Per-worker Workflow Engine. As shown in Figure 10(a), FaaSFlow adopts the Red-Black Deployment to manage different sub-graph versions in worker engines, which is a mechanism supporting FaaS orchestration. It ensures that only the up-to-date version is getting triggered (e.g., sub-graphs with *Version:Y*) at any point in time, while the containers running in out-of-date version (e.g., sub-graphs with *Version:X*) will get recycled once all function tasks return their states. By such means, auto-scaling of function containers in each sub-graph is performed similarly to prior FaaS frameworks. That is, the worker engine pushes the task to a queue for containers to capture, and cold startup happens when there is no warm container for a function invocation.

4.3 Memory Reclaimation in FaaStore

Quota refers to data size limitation in the RPC payload and is used in previous workflow systems. For larger data transfer, functions need remote storage with heavy movement overhead in those systems. FaaSFlow enables large data caching for co-located functions with the in-memory quota of the workflow graph, and specifies the in-memory space exclusive for workflow invocations. *FaaStore* reclaims the memory following the schematic shown in Figure 10(b).

4.3.1 Adaptive In-Memory Storage Quota. Using in-memory storage of *FaaStore* will make the data reside in memory. Although it reduces transmission latencies between functions in the same worker node, temporary data exchange also consumes additional memory resources. A large quota for memory usage will add pressure on the physical node, and even cause memory swap or serious OOM (Out-of-memory) in the worst case. To avoid this issue, *FaaStore* sets a well-organized quota for data movement by memory reclamation from the containers.

Since a function cannot fully utilize the provisioned memory $Mem(v_i)$ in the container, which means that we can use the over-provisioned memory resources for in-memory storage. For a function that uses the memory of size S at most in the history, we reclaim the memory of size $Mem(v_i)$-S-μ from it. The pessimistic reclaim ensures that we will not over-reclaim, and leaves the memory of μ for occasional requirements. Each function node will over-provision $O(v_i)$ for *FaaStore* to reclaim by Equation (1). Equation (2) calculates the in-memory quota by reclaiming memory from all function nodes in the workflow.

$$O(v_i) = max\{[Mem(v_i) - S - \mu], 0\}\overline{Map(v_i)}. \quad (1)$$

$$Quota[G(V,E)] = \sum_{i=1}^{n} O(v_i),\ V = \{v_1, v_2, ..., v_n\}. \quad (2)$$

By such means, *FaaStore* will not take up additional memory space on the host, thus avoiding potential stress.

4.3.2 Memory Reclaimation. Before reclaiming over-provisioned memory from containers, FaaStore first allocates a memory pool with the size of the to-be-reclaimed memory and attaches it with the corresponding *WorkflowID*. Then, the container releases to-be-reclaimed memory by setting an updated *cgroup* memory limit. However, the above mechanism does not apply when adopting MicroVM-based sandboxes for function isolation. The dynamic memory hot-unplugs such as memory-ballon and virtio-mem are not recommended for the large runtime overhead and uncertain stability. Instead, built-in in-memory storage can be distributed among all MicroVMs. Both schemes ensure that the reclaimed memory is exclusive for the corresponding workflow invocations.

5 EVALUATION

We now present the experimental evaluation of our WorkerSP based FaaSFlow and the co-design of *FaaStore* by using both scientific workflows and real-world application benchmarks.

5.1 Experiment Setup

We use 4 scientific workflows from Pegasus [3, 38] and 4 real-world applications as our benchmarks (introduced in Section 2.1). We run the benchmarks on an 8-node cluster. One node is deployed with the Graph Scheduler and generates workflow invocations, and 7 nodes are deployed with distributed workflow engines as the workers to perform the computing. The hardware and software setups are listed in Table 3.

In Section 5.2-5.3, we send invocations with one closed-loop client thread, which ensures that at any given time in the system, only one invocation is being executed in this pattern. When measuring the 99%-ile latencies under the different network bandwidth in Section 5.4, we use the open-loop control to take the queueing time and cold startup effect into considerations. When functions cannot get executed within one minute, it results in execution timeout, and the end-to-end latency is marked the 60s. For co-location evaluation in Section 5.5, we deploy one closed-loop thread client

Table 3: Hardware and software setups in the experiment

	Configuration
Node	CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz Cores: 8, DRAM: 32GB, Disk: 100GB SSD with 3000 IOPS
Software	Operating system: Linux with kernel 4.15.0, Gevent: 21.1.2 Database: Apache/couchdb:3.1.1, Redis:6.2.5 Docker server and client version: 20.10.6 Docker runc version: 1.0.0-rc93, Docker containerd version: 1.4.4
Container	Container runtime: Python-3.7.0, Linux with kernel 4.15.7 Resource limit and Lifetime: 1-core with 256MB, 600s Function container limit: 10 for each function on each node

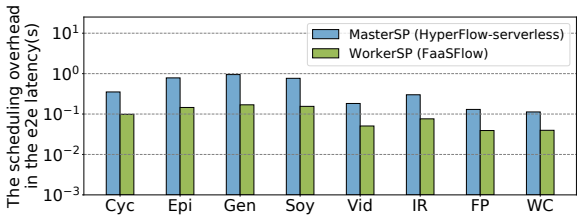


Figure 11: The scheduling overhead of executing workflow benchmarks in both HyperFlow-serverless and FaaSFlow.

for each benchmark and send invocations simultaneously. The measuring still starts at the start of each invocation loop, and ends when receiving the execution state. It ensures that the potential cold startup will not impact the queuing latency, thus avoiding a cascading effect on the end-to-end latency.

In FaaSFlow, we use CouchDB [4] as the remote Key-Value Store and deploy the Redis [50] as in-memory storage. The APIs for both methods are integrated into the FaaSFlow of each node, while the low-level APIs are available for user code to make data transmission. When routing function nodes to different worker nodes, we also modify the routing policy in HyperFlow-serverless to the same way as in FaaSFlow, which satisfies the control variate method.

5.2 Scheduling Overhead

Firstly, we evaluate the impact on scheduling overhead of WorkerSP and traditional MasterSP when workflows are invoked. In this experiment, we send invocations for each benchmark in both HyperFlow-serverless and FaaSFlow. The scheduling overhead is calculated as illustrated in Section 2.3. To avoid randomness, we report the average results in 1000 invocations and show the efficiency for workflows executing in WorkerSP based architecture in Figure 11.

As observed, all the benchmarks have the shortest scheduling overhead and end-to-end response latencies with FaaSFlow. Compared with HyperFlow-serverless, 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. The scheduling overhead in scientific workflows in higher than real-world benchmarks results from more function nodes and branches in the former. However, the optimization is effective no matter how many function nodes a workflow has. Furthermore, task assignment through TCP connections between the master and workers is fully eliminated. Only state transitions are left, making FaaSFlow require less scheduling overhead than MasterSP based HyperFlow-serverless.

Table 4: The overall latencies of data movement in all edges of the benchmarks.

Latency(s)	Cyc	Epi	Gen	Soy	Vid	IR	FP	WC
HyperFlow -serverless	204.2	2.23	29.26	10.06	4.02	0.20	1.29	1.46
FaaSFlow -FaaStore	10.28	0.69	22.17	9.53	1.03	0.13	0.49	0.21
Reduced transmission	95%	69%	24%	5.2%	74%	35%	62%	70%

5.3 Overhead of Data Movement

Besides evaluating the end-to-end latencies of scheduling overhead for different benchmarks, we assess the effectiveness of reducing the transmission overhead with FaaSFlow. By leveraging the proposed adaptive storage library *FaaStore*, FaaSFlow significantly alleviates the data movement and reduces the end-to-end latency. We denote the FaaSFlow with *FaaStore* activated as FaaSFlow-FaaStore, and compare its end-to-end performance with HyperFlow-serverless.

Table 4 shows the data movement overhead of HyperFlow-serverless and FaaSFlow-FasStore (the size of transferred data is shown in Figure 5). It should be noticed that the data movement overhead represents all the data transmission in the DAG rather than in the critical path. It explains why the overall data movement latencies for HyperFlow-serverless are much higher than the end-to-end latencies. The key is that the distributed workflow engine can be aware of the function and its successor functions' location, thus alleviating the unnecessary database storage for temporary data by localizing them into memory. Meanwhile, FaaSFlow-FaaStore does not allocate additional memory space for the localized data but makes the memory reclamation that is over-subscribed by functions in containers.

5.4 Tail Latency and Throughput

This experiment shows the throughput improvement of the benchmarks in FaaSFlow-FaaStore compared with HyperFlow-serverless. In this experiment, we record the 99%-ile latency by sending invocations for each benchmark at different invocation rates. In each test, we run the benchmark for 1000 invocations. As an example, Figure 13 shows the 99%-ile latencies of the benchmarks when they are invoked 6 invocations per minute. In the figure, the x-axis shows the load of the benchmark. HyperFlow-serverless can handle most benchmarks with a 50MB/s network bandwidth. Compared with HyperFlow-serverless, FaaSFlow-FaaStore further reduces the 99%-ile latencies of the benchmarks *Epi, Soy, Vid, IR, FP*, and *WC* by 23.3% on average, and reduces the 99%-ile latencies of the benchmarks *Cyc* and *Gen* by 75.2% on average.

In Figure 13, if a bar is as high as 60s, the function is not invoked (execution timeout) in the current network and load configuration. The timeout happens for *Gen*, and *Cyc*, because the network bottleneck exists in the parallel and foreach steps. In this case, a large number of function instances contend for the limited network

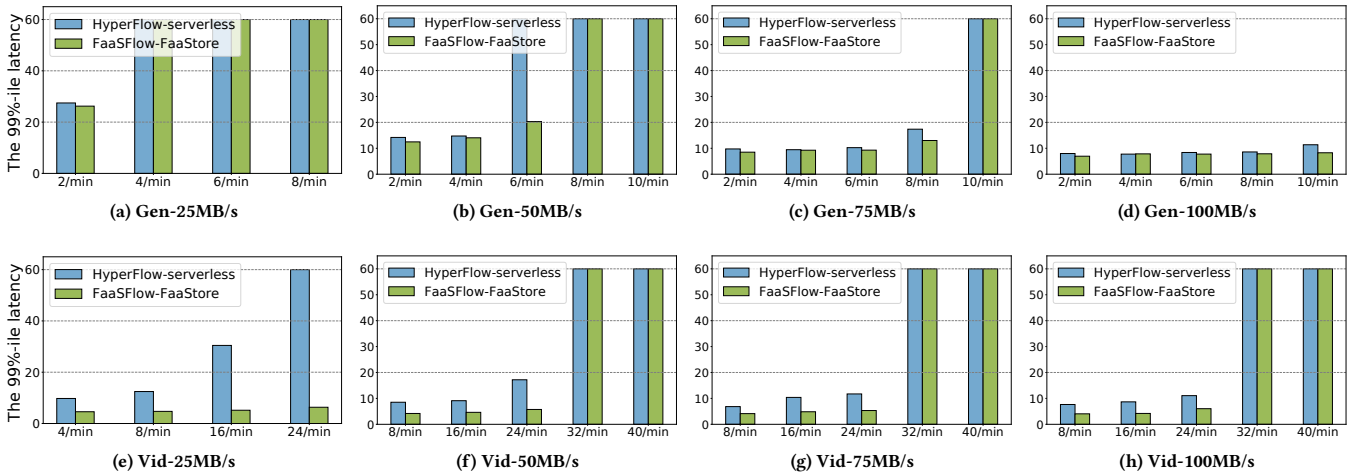


Figure 12: The 99%-ile latencies under different thourghout for *Gen* and *Vid* when the network bandwidth of the storage node is configured with 25MB/s, 50MB/s, 75MB/s, and 100MB/s.

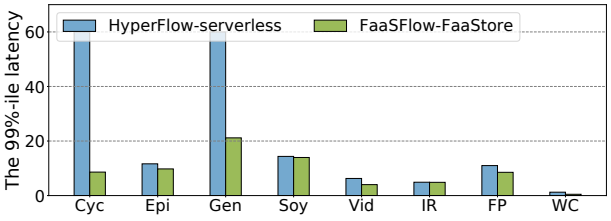


Figure 13: The 99%-ile e2e latency when benchmarks running under a 50MB/s bandwidth setup (by sending 6 invocations per minute for each one).

bandwidth. By enabling the data transfer on the local node, the network bottleneck is alleviated with FaaSFlow-FaaStore. Therefore, the 99%-ile latencies of the benchmarks are reduced.

Due to the limited space, we use *Gen* and *Vid* to evaluate the performance of FaaSFlow-FaaStore in reducing the tail latency and improving the throughput with different network bandwidth configurations. Other benchmarks show similar results. The experimental results are shown in Figure 12. As observed, HyperFlow-serverless improves the throughputs of the benchmarks compared with HyperFlow-serverless when the network bandwidth is larger. This is because both the two benchmarks show heavy data transmission between functions. The performance is poor when the network bandwidth is small, because the functions contend for the limited network bandwidth with HyperFlow-serverless.

Observed from Figure 12, the data-intensive benchmarks with HyperFlow-serverless is highly sensitive to the network bandwidth. On the contrary, FaaSFlow-FaaStore mitigates the performance degradation of the workflow benchmarks when the bandwidth is small. This is because most of the data transmission is done locally via FaaSFlow-FaaStore. We can also observe that the 99%-ile latencies of *Gen* and *Vid* with FaaSFlow-FaaStore when the network bandwidth is 50MB/s and 25MB/s are similar to their 99%-ile latencies with HyperFlow-serverless when the network bandwidth 75MB/s and 100MB/s, respectively. In this case, if the network bandwidth drops to 25MB/s because other workloads take most of the bandwidth, the benchmarks with Hyperflow-serverless suffers from 32.5% throughput degradation on average. On the contrary, the degradation of FaaSFlow-FaaStore is smaller than 9.5%.

To conclude, network bandwidth utilization can be increased by 1.5X-4X when localizing the temporary data between functions into the local node with FaaSFlow-FaaStore.

5.5 Co-location Interference

We also focus on the performance when multiple workflows co-run in the same cluster to evaluate the degradation when making each benchmark running from a solo-run mode to a co-run manner. In this experiment, we follow the hardware and software setup introduced in Section 5.1, but run all these 8 benchmarks on the cluster simultaneously. Each workflow benchmark client still sends invocations in a closed-loop pattern. We report the end-to-end latencies for HyperFlow-serverless based deployment and FaaSFlow-FaaStore based deployment in Figure 14.

When benchmarks are co-running in HyperFlow-serverless, they will contend for the shared bandwidth resources, especially for workflow benchmarks with numerous parallel and foreach steps. For example, benchmarks of *Cyc*, *Gen*, *Vid*, *WC* have a significant performance degradation of 50.3%, 48.5%, 84.4%, 66.2%, respectively, when co-running in HyperFlow-serverless. However, FaaSFlow-FaaStore can effectively alleviate the degradation when workflows are co-located. We find it is because, in FaaSFlow-FaaStore, function nodes in these benchmarks will be well-proportioned distributed across all nodes after Graph Scheduler. In this case, each worker node can localize temporary data, therefore alleviating the bandwidth bottleneck. To validate the effectiveness of the Graph Scheduler, we also show the grouping and scheduling result for 8 benchmarks in Figure 15.

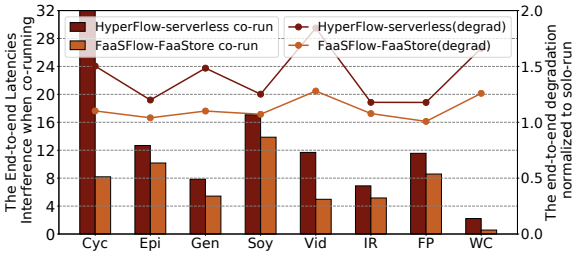


Figure 14: The co-location interference in end-to-end latencies of 8 benchmarks when executing them in HyperFlow-serverless and FaaSFlow-FaaStore.

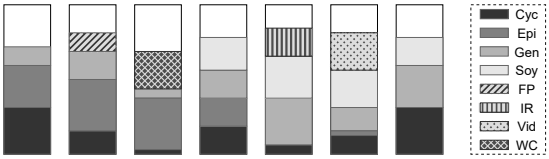


Figure 15: The scheduling result and distribution.

As shown in the figure, all scientific workflow benchmarks with 50 function nodes will get distributed across the 7 worker nodes, while real-world benchmarks with around 10 functions nodes or less will be grouped and scheduled to the same worker node. It is because function nodes with heavy data movement are more likely to be partitioned into more groups, and function nodes with less data movement will be scheduled to balance the load and resource.

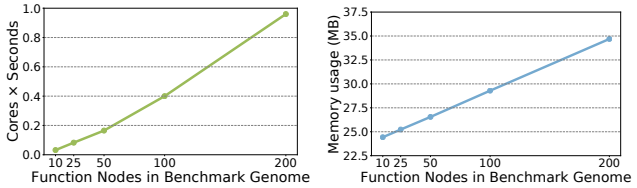
5.6 Scalability of the Scheduler

Current serverless systems require both low response latency and high availability. The overhead of task scheduler in making decisions and scheduling is critical for cloud orchestrators. To this end, we aim to evaluate the performance of the dynamic scheduling in Graph Scheduler. We use Genome as the benchmark to investigate workflows' grouping and scheduling performance because it provides scalability to change the number of nodes it contains. By changing the number of function nodes to 10, 25, 50, 100, 200, respectively, we time the scheduler response and show the results in Figure 16.

According to our analysis, with the increase of the number of function nodes, the overhead roughly follows the time complexity of $O(n^2)$. We also evaluate the resources used when the number of worker nodes increases, the average CPU and memory usage keep stable. We find that the memory usage of the scheduler starts from 24.43MB for that all the associated component overhead and data metric are also included. Our Graph Scheduler shows greater performance on scheduling for most current internet applications with less than 50 function nodes.

5.7 FaaSFlow Components Overhead

Besides Graph Scheduler, we also focus on the CPU and memory overhead introduced by Workflow Engine. Experiment results show



(a) CPU cores (b) Memory

Figure 16: The average CPU and memory usage for Graph Scheduler when invoke workflow benchmarks with different number of function nodes.

that the average CPU and memory usage for Workflow Engine deployment are 0.12 core and 47MB, respectively, in each worker node. In general, the CPU and memory overhead of running the distributed engines in the cluster is acceptable so that more resources can be used to handle workflow invocations. Furthermore, by runtime recycling of the memory invocations, less memory consumption is required for each workflow.

We also run FaaSFlow with the different number of worker nodes (ranging from 1 to 100) to simulate various cluster scenarios and monitor the CPU and memory usage when the cluster is processing multiple workflows invocations. In this experiment, workflow invocations are also sent in a closed-loop pattern. According to the result, the average CPU and memory usage scale linearly as the number of nodes increases, and it does not introduce additional CPU or memory overhead to execute a workflow invocation. In conclusion, FaaSFlow does not introduce extra overhead for workflow invocations when scaling up the cluster.

6 IMPLICATIONS FOR SERVERLESS CLOUDS

According to our experiment, we have two key implications.

More server nodes can be integrated in a serverless cloud with the worker-side workflow schedule pattern. While more and more servers are integrated into the cloud, the traditional MasterSP fails to support too many servers, as the master node has to trigger the functions for all the worker nodes. By adopting the proposed WorkerSP, the master node only needs to split and distribute sub workflows to the functions. In this case, the scheduling overhead of tasks assigning through TCP connections is alleviated by locally triggering functions on each node.

Deploying servers with larger main memory is more beneficial than upgrading the network for serverless workflows. More intermediate data can be stored when the main memory is larger. Localized data transmission helps multiple the utilization of network bandwidth and reduces throughput degradation. As stated before, FaaSFlow-FaaStore is able to increase the network bandwidth utilization by up to 1.5X or 4X through local data transmission.

7 RELATED WORK

Cold startups and prewarm strategies. In serverless computing or FaaS-based workflows, it is a critical problem that functions need to experience the cold startup for the first invocation, which has

been a significant topic for researchers to reduce the performance degradation due to cold startups [29, 39, 46, 47, 53, 58]. Shahrad *et al.* [53] propose a dynamic resource management policy for the private prewarmed container pool. By pausing containers according to the time series prediction, it reloads before invocations arrive with the cold startup. Catalyzer [29] leverages a template sandbox that already has pre-loaded the specific function for state resuing. They propose the sandbox fork by pre-loading the critical path files of the image checkpoint to accelerate the container recovery. SOCK [47] uses the generic Zygote mechanism that caches the most benefit-to-cost packages into the common prewarmed Zygotes. The lean SOCK containers integrated with the tree-based caching show better performance in reducing the cold startups. FaasCache [32] considers that keeping a function warm is equivalent to caching objects, and therefore implements the Greedy-Dual keep-alive caching to reduce the resource requirement and cold startups.

However, these studies only focus on the function-level rather than the application-level optimization, still making serverless workflows suffer from huge scheduling and transmission overhead.

Serverless workflow optimizations. Currently, the majority of serverless workflow engines or frameworks adopts the Master-Worker approach by identifying the state of ready functions in the master node and assign tasks [20, 21, 30, 42, 43], including AWS Steps Functions [6] and Fission Workflows [7]. Prior researchers have also investigated that overhead for parallelism grows exponentially with the number of parallel functions in cloud serverless workflow platforms [36, 41, 54]. In this case, workflows in a serverless context require more attention. WuKong [24] implement a publisher/subscriber-based serverless parallel framework. It performs task scheduling by a fleet of lambda executors and partition a DAG into multiple subgraphs. Intermediate task inputs and outputs may be stored inside the same executors to enhance the data locality. Viil *et al.* [59] share a similar idea to use the task partitioning and migration to provision and configure the resources automatically. NightCore [35] makes the assumption that all functions of the application can be scheduled on the same server, and it makes sense by reducing RPC overheads for functions on the same node. However, Nightcore falls back to the MasterSP between different worker servers, because their gateway serves as the centralized manager for assigning tasks and allocating execution states.

Compared with our approach of triggering in each worker node, these solutions are still MasterSP-based approaches that cannot provide efficient scheduling and data locality. In constant, FaaSFlow eliminates the task assigning step of the workflow engine, and cross-server scheduling overhead is greatly reduced.

8 CONCLUSION

In this paper, we propose FaaSFlow, a master-worker computing architecture-based serverless system for workflow executions. It distributes workflow engines in each worker node and makes the function node triggered locally to reduce the scheduling overhead by networks. It also groups the function nodes into a subgraph and enables the direct data movement through in-memory storage to enhance the data locality. Experiment results show that both scientific workflows and real-world applications can significantly reduce the scheduling and transmission overhead in FaaSFlow.

ACKNOWLEDGMENT

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006), and Shanghai international science and technology collaboration project (No.21510713600). Quan Chen and Minyi Guo are the corresponding authors.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact includes the prototype implementation of MasterSP and WorkerSP in FaaSFlow, the 8 workflow benchmarks evaluated in our experiments, and some experiment workflow scripts to reproduce our results on Alibaba ECS instances.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** A grouping algorithm that is used in Graph Scheduler to partition sub-graphs and make scheduling decisions.
- **Programs:** FaaSFlow, Real-world application benchmarks, Docker runtime, wondershaper, CouchDB, and Redis.
- **Data set:** Scientific workflow traces generated from Pegasus workflow executions, are included in the artifact.
- **Run-time environment:** Alibaba ECS instance (ecs.g7.2xlarge) for each worker node, Alibaba ECS instance (ecs.g6e.4xlarge) for the storage node. Detailed software libraries (including docker runtime, CouchDB, Redis, and python packages) are all listed and scripted in the artifact.
- **Hardware:** For each ecs.g7.2xlarge instance, the hardware is configured by {CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz, Cores: 8, DRAM: 32GB, Disk: 100GB SSD with 3000 IOPS.} For the ecs.g6e.4xlarge instance, the hardware is configured by {CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz, Cores: 16, DRAM: 64GB, Disk: 100GB SSD with 3000 IOPS.}
- **Execution:** For reproducing our paper's results, we provide the corresponding scripts for each evaluation section (from Section 5.2 to Section 5.7) to send queries, collect the execution metrics, and draw the comparison figures just like ours.
- **Metrics:** Metrics are collected in each partitioning iteration, including end-to-end scheduling overhead (Section 5.2), data movement latencies (Section 5.3), end-to-end 99%-ile latencies (Section 5.4), co-location degradations (Section 5.5), and CPU/Memory usage (Section 5.6 and Section 5.7).
- **Output:** Scripts for reproducing results in each experiment evaluation are included in the artifact.
- **Time needed to prepare workflow:** 1 hour
- **Time needed to complete experiments:** 5-6 hours
- **Publicly available:** Yes
- **Archived:** 10.5281/zenodo.5900766

A.3 Description

A.3.1 How to Access. The source code of FaaSFlow and benchmarks are available and maintained on GitHub lzjzx1122/FaaSFlow.

A.3.2 Hardware Dependencies. We recommend to use Alibaba ECS instance (ecs.g7.2xlarge) for each worker node, and Alibaba ECS instance (ecs.g6e.4xlarge) for the storage node. Each ecs.c6e.2xlarge instance is configured by {CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz, Cores: 8, DRAM: 32GB, Disk: 100GB SSD with 3000 IOPS}. For the ecs.g6e.4xlarge instance, the hardware is configured by {CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz, Cores: 16,

DRAM: 64GB, Disk: 100GB SSD with 3000 IOPS.} When performing the evaluation in Section 5.4, the Network Bandwidth setup should be limited in storage node from 25MB/s to 100MB/s by `wondershaper`.

A.3.3 Software Dependencies. The artifact requires experiment VMs running Ubuntu 18.04 with Docker installed. We provide an installation script to prepare the software environment (including docker runtime, CouchDB, Redis, and python packages) for both worker node and storage node.

A.4 Installation

To reproduce our experiment, we recommend building an 8-nodes cluster, where 7 nodes are used for performing the function execution, and 1 node is used for remote storage and queries generating. `README.md` of the artifact provides more details on reproducing our results of FaaSFlow.

Setting up the storage node. For the storage node, reset configuration of `worker_address` in `src/grouping/node_info.yaml`. It will specify your workers' addresses. The default configuration of `scale_limit` represents the maximum container numbers that can be deployed in each 32GB memory instance, and it does not need any change by default. After that, run the script `scripts/db_setup.bash` to prepare runtime dependencies, including docker, CouchDB, required python packages, and build grouping results for 8 benchmarks.

Setting up the worker node. For each worker, reset configuration of `COUCHDB_URL` in `src/container/container_config.py` and `config/config.py`. The url should be set to the corresponding DB in the storage node you build previously. Run the script `scripts/worker_setup.bash` to prepare runtime dependencies, including docker, redis, required python packages, and build docker images for 8 benchmarks.

MasterSP and WorkerSP. MasterSP and WorkerSP are all included in our artifact. `DATA_MODE` indicates whether FaaStore is enabled in each worker node, while `CONTROL_MODE` indicates whether functions are local triggered (WorkerSP) or assigned from the master scheduler (MasterSP). If MaterSP is enabled, you should start another proxy at any worker node as the master node, and export the corresponding IP and port in `MASTER_HOST`. For more commands details, please check the `README.md` in our artifact.

A.5 Experiment Workflow

Each sub-directory within `test/asplos` corresponds to one experiment. Within each directory, the entry point of each experiment is the `run.py` script. Detailed scripts usage is introduced in the `README.md` of our artifact. It first generates the benchmarks invocations and sends them to worker nodes in either WorkerSP or MasterSP mode. Then it collects the execution metrics and stores them in a `.csv` or `.txt` file for this experiment.

A.6 Evaluation and Expected Results

We provide a helper script `draw.sh` in the `expected_result` to make the comparison figures. The expected results are execution logs from our experiment runs, and stored in the `expected_result` directory within each experiment subdirectory. They include end-to-end scheduling overhead (Section 5.2), data movement latencies (Section 5.3), end-to-end 99%-ile latencies (Section 5.4), co-location degradations (Section 5.5), and CPU/Memory usage (Section 5.6 and Section 5.7).

REFERENCES

[1] 2021. Alibaba Serverless Workflow: Visualization, free orchestration, and Coordination of Stateful Application Scenarios. https://www.alibabacloud.com/product/serverless-workflow.

[2] 2021. AWS Step Functions: assemble functions into business-critical applications. https://aws.amazon.com/step-functions/.

[3] 2021. Collection of workflow execution instances for the Pegasus workflow management system. https://github.com/wfcommons/pegasus-instances.

[4] 2021. CouchDB. https://couchdb.apache.org/.

[5] 2021. Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless compute environment. https://docs.microsoft.com/en-us/azure/azure-functions/durable/.

[6] 2021. Elastic Load Balancing Application Load Balancers. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/elb-ag.pdf.

[7] 2021. Fission Workflows: Fast, reliable and lightweight function composition for serverless functions. https://github.com/fission/fission-workflows.

[8] 2021. Google Cloud Functions: Scalable pay-as-you-go functions as a service (FaaS) to run your code with zero server management. https://cloud.google.com/functions/.

[9] 2021. OpenWhisk: Serverless functions platform for building cloud applications. https://github.com/apache/openwhisk.

[10] 2021. Quota of AWS Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/limits-overview.html.

[11] 2021. Quota of Google Cloud Functions. https://cloud.google.com/workflows/quotas.

[12] 2021. Quota of microsoft Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#service-limits.

[13] 2021. Serverless Community Survey: huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage.

[14] 2021. Serverless Reference Architecture for Real-time File Processing. https://github.com/aws-samples/lambda-refarch-fileprocessing.

[15] 2021. Tutorial for detecting and bluring offensive images using Cloud Functions. https://cloud.google.com/functions/docs/tutorials/imagemagick.

[16] 2021. Tutorial for Perfoming Optical Character Recognition (OCR) on Google Cloud Platform. https://cloud.google.com/functions/docs/tutorials/ocr.

[17] 2021. Use FFmpeg in Function Compute to process audio and video files in Function Compute. https://www.alibabacloud.com/help/doc-detail/146712.htm?spm=a2c63.l28256.b99.313.5c293c94dPLJV1.

[18] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. 2019. A Survey on Scheduling Strategies for Workflows in Cloud Environment and Emerging Trends. *ACM Comput. Surv.* 52, 4 (2019), 68:1–68:36. https://doi.org/10.1145/3325097

[19] Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 884–889. https://doi.org/10.1145/3106237.3117767

[20] 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, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 263–274. https://doi.org/10.1145/3267809.3267815

[21] Bartosz Balis. 2016. HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows. *Future Gener. Comput. Syst.* 55 (2016), 147–162. https://doi.org/10.1016/j.future.2015.08.015

[22] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. 2012. Bi-criteria Workflow Tasks Allocation and Scheduling in Cloud Computing Environments. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*. IEEE Computer Society, 638–645. https://doi.org/10.1109/CLOUD.2012.83

[23] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.* 25, 6 (2009), 599–616. https://doi.org/10.1016/j.future.2008.12.001

[24] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 1–15. https://doi.org/10.1145/3419111.3421286

[25] Quan Chen, Shuai Xue, Shang Zhao, Shanpei Chen, Yihao Wu, Yu Xu, Zhuo Song, Tao Ma, Yong Yang, and Minyi Guo. 2020. Alita: comprehensive performance isolation through bias resource management for public clouds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage*

and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 32. https://doi.org/10.1109/SC41405.2020.00036

[26] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 17–32. https://doi.org/10.1145/3037697.3037700

[27] Rafael Ferreira da Silva, Rosa Filgueira, Ewa Deelman, Erola Pairo-Castineira, Ian Michael Overton, and Malcolm P. Atkinson. 2019. Using simple PID-inspired controllers for online resilient resource management of distributed scientific workflows. *Future Gener. Comput. Syst.* 95 (2019), 615–628. https://doi.org/10.1016/j.future.2019.01.015

[28] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35. https://doi.org/10.1016/j.future.2014.10.008

[29] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[30] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[31] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. 2021. QoS-Aware and Resource Efficient Microservice Deployment in Cloud-Edge Continuum. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 932–941. https://doi.org/10.1109/IPDPS49936.2021.00102

[32] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery Berger, and Christos Kozyrakis (Eds.). ACM, 386–400. https://doi.org/10.1145/3445814.3446757

[33] Siqian Gong, Beibei Yin, and Kai-Yuan Cai. 2018. An Adaptive PID Control for QoS Management in Cloud Computing System. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, USA, October 15-18, 2018*, Sudipto Ghosh, Roberto Natella, Bojan Cukic, Robin S. Poston, and Nuno Laranjeiro (Eds.). IEEE Computer Society, 142–143. https://doi.org/10.1109/ISSREW.2018.00-12

[34] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[35] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 152–166. https://doi.org/10.1145/3445814.3446701

[36] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 445–451. https://doi.org/10.1145/3127479.3128601

[37] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[38] Gideon Juve, Ann L. Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. 2013. Characterizing and profiling scientific workflows. *Future Gener. Comput. Syst.* 29, 3 (2013), 682–692. https://doi.org/10.1016/j.future.2012.08.015

[39] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. 2021. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.* (dec 2021). https://doi.org/10.1145/3508360 Just Accepted.

[40] Changyuan Lin and Hamzeh Khazaei. 2021. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 615–632. https://doi.org/10.1109/TPDS.2020.3028841

[41] Pedro García López, Marc Sánchez Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. 2018. Comparison of FaaS Orchestration Systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018, Zurich, Switzerland, December 17-20, 2018*, Alan Sill and Josef Spillner (Eds.). IEEE, 148–153. https://doi.org/10.1109/UCC-Companion.2018.00049

[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 *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[43] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2020. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Gener. Comput. Syst.* 110 (2020), 502–514. https://doi.org/10.1016/j.future.2017.10.029

[44] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, Carlo Galuzzi, Luigi Carro, Andreas Moshovos, and Milos Prvulovic (Eds.). ACM, 248–259. https://doi.org/10.1145/2155620.2155650

[45] Mohammad Masdari, Sima ValiKardan, Zahra Shahi, and Sonay Imani Azar. 2016. Towards workflow scheduling in cloud computing: A comprehensive analysis. *J. Netw. Comput. Appl.* 66 (2016), 64–82. https://doi.org/10.1016/j.jnca.2016.01.018

[46] M. Garrett McGrath and Paul R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino (Eds.). IEEE Computer Society, 405–410. https://doi.org/10.1109/ICDCSW.2017.36

[47] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[48] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 69–84. https://doi.org/10.1145/2517349.2522716

[49] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[50] Salvatore Sanfilippo. 2021. Redis: Remote Dictionary Server. https://redis.io/.

[51] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). ACM, 351–364. https://doi.org/10.1145/2465351.2465386

[52] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 1063–1075. https://doi.org/10.1145/3352460.3358296

[53] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, and Gohar Chaudhry. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[54] Vaishaal Shankar, Karl Krauth, and Qifan Pu. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018). arXiv:1810.09679 http://arxiv.org/abs/1810.09679

[55] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2020. SNF: serverless network functions. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 296–310. https://doi.org/10.1145/3419111.3421295

[56] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, and Johann Schleier-Smith. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13,

11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf

[57] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: enabling quality-of-service in serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 311–327. https://doi.org/10.1145/3419111.3421306

[58] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 559–572. https://doi.org/10.1145/3445814.3446714

[59] Jaagup Viil and Satish Narayana Srirama. 2018. Framework for automated partitioning and execution of scientific workflows in the cloud. *J. Supercomput.* 74, 6 (2018), 2656–2683. https://doi.org/10.1007/s11227-018-2296-7

[60] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *ATC*. 133–146.

[61] Philipp A. Witte, Mathias Louboutin, Henryk Modzelewski, Charles Jones, James Selvage, and Felix J. Herrmann. 2020. An Event-Driven Approach to Serverless Seismic Imaging in the Cloud. *IEEE Trans. Parallel Distributed Syst.* 31, 9 (2020), 2032–2049. https://doi.org/10.1109/TPDS.2020.2982626

[62] Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, Avi Mendelson (Ed.). ACM, 607–618. https://doi.org/10.1145/2485922.2485974

[63] Tianyi Yu, Qingyuan Liu, and Dong Du. 2020. Characterizing serverless platforms with serverlessbench. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 30–44. https://doi.org/10.1145/3419111.3421280

[64] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 659–670. https://doi.org/10.1109/ICDE.2017.119

[65] Ge Zheng and Yang Peng. 2019. GlobalFlow: A Cross-Region Orchestration Service for Serverless Computing Services. In *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, Elisa Bertino, Carl K. Chang, and Peter Chen (Eds.). IEEE, 508–510. https://doi.org/10.1109/CLOUD.2019.00093