Journal of Parallel and Distributed Computing 167 (2022) 18–30

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Reducing response latency of composite functions-as-a-service through scheduling

Pawel Zuk *, Krzysztof Rzadca

Institute of Informatics, Banacha 2, 02-097, Warsaw, Poland

ARTICLE INFO

Article history:
Received 27 February 2021
Received in revised form 11 February 2022
Accepted 9 April 2022
Available online 19 April 2022

Keywords:
Scheduling
Workflow
DAG
Setup times
Serverless

ABSTRACT

In Function-as-a-Service (FaaS) clouds, customers deploy to cloud individual functions, in contrast to complete virtual machines (IaaS) or Linux containers (PaaS). FaaS offerings are available in the largest public clouds (Amazon Lambda, Google Cloud Functions, Azure Serverless); there are also popular open-source implementations (Apache OpenWhisk) with commercial offerings (Adobe I/O Runtime, IBM Cloud Functions). A recent addition to FaaS is the ability to compose functions: a function may call another functions, which, in turn, may call yet another function — forming a directed acyclic graph (DAG) of invocations. From the perspective of the infrastructure, a composed function is less opaque than a virtual machine or a container. We show that this additional information about the internal structure of the function enables the infrastructure provider to reduce the response latency. In particular, knowing the successors of a function in a DAG, the infrastructure can schedule these future invocations along with necessary preparation of environments.

We model resource management in FaaS as a scheduling problem combining (1) sequencing of invocations; (2) deploying execution environments on machines; and (3) allocating invocations to deployed environments. For each aspect, we propose heuristics that employ FaaS-specific features. We explore their performance by simulation on a range of synthetic workloads and on workloads inspired by trace from existing system. Our results show that if the setup times are long compared to invocation times, algorithms that use information about the composition of functions consistently outperform greedy, myopic algorithms, leading to significant decrease in response latency.

1. Introduction

Serverless computing allows a cloud customer to run their code in production without configuring and allocating the software and the infrastructure stack [8]. A cloud customer can thus focus on their application, rather than on managing the production environment. Major cloud providers offer serverless products (Amazon Lambda, Google Cloud Functions, Microsoft Azure Serverless). We focus on a variant of serverless computing called *Function as a Service (FaaS)* [12]. In FaaS, a cloud customer uploads the source code of a (stateless) function to the provider. When an end-user issues a request, this code is executed on the infrastructure provided and managed by the FaaS system. The FaaS system isolates requests by providing for each invocation a prepared execution environment — usually a Linux container.[1]

While FaaS is gaining wide adoption, a recent new element is still relatively unexplored — the *composition* of functions [4]. During an invocation of a composed FaaS initiated by a single incoming event (e.g., an HTTP request), a function calls another function, that, in turn, may call yet another function and so on. If these invocations are all synchronous, the call structure is a chain; if some are asynchronous, it is a DAG. In this paper, we focus on DAGs (our conference paper [40] studied chains — in this paper we extend our algorithms to DAGs and we also compare how the additional complexity of DAGs influence the results).

The existing open-source FaaS systems (OpenWhisk, Fission Workflows) do not use the information about the structure of

* Corresponding author.
E-mail addresses: p.zuk@mimuw.edu.pl (P. Zuk), krzadca@mimuw.edu.pl (K. Rzadca).

https://doi.org/10.1016/j.jpdc.2022.04.011
0743-7315/

[1] This is an extended and updated version of our conference paper: Pawel Zuk and Krzysztof Rzadca, Scheduling Methods to Reduce Response Latency of Function as a Service, IEEE 32nd International Symposium on Computer Architecture and High Performance Computing, Porto, Portugal, 2020.

the function compositions. Each invocation in a composition (in a chain or a DAG) is treated independently. However, once the first function is invoked, the scheduler knows that the functions that follow in a DAG will be eventually called too — thus, the scheduler can prepare their execution environments in advance. Moreover, information about current system state can be used to perform optimizations by changing order of execution of incoming invocations.

The contributions of this paper are as follows:

- We model scheduling in FaaS as a combination of the multiple knapsack problem, scheduling with dependencies and with setup times (Section 2).
- We propose a number of heuristics for each aspect (Section 3). These heuristics derive from classic approaches, but we adjust them to the FaaS specificity.
- By simulations, we show that heuristics examining the composition structure lead to lower response latencies (Section 4).

2. Modeling FaaS resource management

2.1. Resource management in OpenWhisk

In this section, we describe from the resource management perspective a representative implementation of a serverless cloud platform, the open-source Apache OpenWhisk [3]. OpenWhisk is mature, actively-developed software also offered commercially (IBM Cloud Functions, Adobe I/O Runtime). OpenWhisk alternatives include OpenLambda [17] and Fission [20]. OpenLambda uses containers to provide runtime environment for functions. Fission is designed for Kubernetes [21]; it can be deployed on existing cluster among other applications, which makes its adoption significantly easier. This section forms a background for our scheduling model that follows in Section 2.2.

OpenWhisk allows a cloud *customer* to upload *functions* (essentially, code snippets). A function is executed when *end-users* issue requests. A function executes in an *environment* — an initialized Linux container. Different container images are used for each of supported languages; a customer can also provide a custom image (with, e.g., additional libraries). Before the first execution of a function, the container must be initialized (e.g., setting up the container or compiling a Go function code). This initialization can take a considerable amount of time (called later the *setup time*) — [25] reports at least 500 ms. An environment is specific to a function — an environment is not reused by different functions. However, subsequent invocations may reuse the same environment without the need to re-initialize it, thus, without the increased latency caused by the setup time. By default, in OpenWhisk each environment executes at most a single invocation at any given moment (there is no parallelism inside an environment). However, multiple independent invocations can be processed in parallel by multiple environments.

OpenWhisk also allows to compose several functions into a chain (a sequence). After one function finishes, its result are passed to the next function; the last function responds to the end-user. While sequences are natively supported, in order to spawn two or more functions in parallel (resulting in a DAG), the developer may use an additional OpenWhisk Composer module or call the OpenWhisk API from the function code. These composed functions are now relatively uncommon. To the best of our knowledge, there is no publicly available FaaS trace with function composition and setup times. However, we argue that their introduction follows the standard trend in software engineering of refactoring large functions into a series of smaller ones; or from monolith applications to meshes of microservices. FaaS is still a new paradigm and we assume that soon this trend will follow.

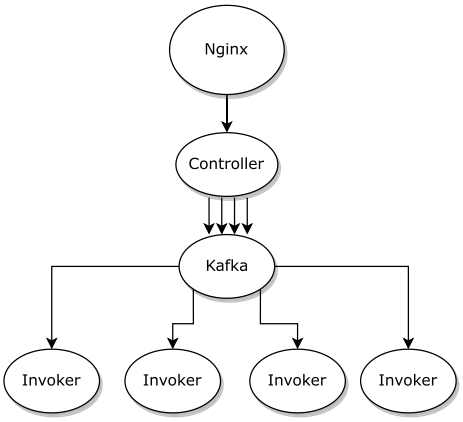


Fig. 1. Core architecture of OpenWhisk.

Architecture of OpenWhisk is complex. Fig. 1 presents a high-level overview of OpenWhisk internal modules. From our perspective the key components are the *controller* and the *invoker*. The controller communicates with the invokers by message passing (via Apache Kafka).

The invoker is an agent running on a worker node. The invoker is responsible for executing actions scheduled on a particular node. Each invoker has a unique identifier; it announces itself to the controller while starting.

The controller acts as a scheduler handling incoming events and routing each function invocation to a concrete invoker. The controller monitors the status of workers and the currently executing invocations.

The controller attempts to balance load across nodes. The default algorithm selects for each function the *initial worker node* based on a hash of the workspace name and the function name. Similarly, the algorithm picks for each function another number, called the *step size* (a number co-prime with the count of worker nodes). Each time a function is invoked, the controller attempts to schedule the invocation on its initial worker. If a worker doesn't have sufficient resources immediately available, the controller tries to schedule the invocation on the next node (increased by the *step size*). If the invocation cannot be immediately scheduled on any node, it is queued on a randomly chosen node.

2.2. A scheduling model for FaaS

In this section we define the optimization model for the FaaS resource management problem. The aim of this model is to have the simplest possible (yet still realistic) approximation of a FaaS system that enables us to show that explicitly considering FaaS compositions allows optimizations. We thus deliberately do not take into account some factors that we argue are orthogonal for this work.

We use the standard notation from [6]. A single end-user request corresponds to a *job* J_i. A job is composed of one or more *tasks* $O_{i,k}$, each corresponding to a single FaaS invocation. The request is responded to (the job completes) at time C_i when the last task completes, $C_i = \max_k C_{i,k}$ (where $C_{i,k}$ denotes the completion time of task $O_{i,k}$). Tasks have dependencies resulting from, e.g., before-after relationships in the code. We denote set of task $O_{i,k}$ dependencies (predecessors) by $P_{i,k}$, i.e., task $O_{i,k}$ may start (at time $\sigma_{i,k}$) only after all its predecessors $\forall_{j \in Pi,k} O_{i,j}$ complete, $\sigma_{i,k} \geq \max_{j \in P_{i,k}} C_{i,j}$.

We assume that individual functions are repeatedly executed (modeling similar requests from many end-users but also shared modules like authorization). We model such grouping by mapping each task $O_{i,k}$ to exactly one *task family* $f(O_{i,k})$ (obviously, two

tasks $O_{i,k}$ and $O_{i,l}$ from a job J_i might belong to different families). All tasks from a family f require the same environment E_f, have the same execution time (duration) p_f and require the same amount of resources q_f.

A task $O_{i,k}$ from a family $f(O_{i,k})$ is executed on exactly one machine in an *environment* (OS container) E_f. E_f requires set-up time s_f (initialization of the environment) before executing the first task. Subsequent tasks executed in this environment do not require set-up times. Typically, s_f is non-negligible and much longer than the task's duration, $s_f > p_f$ but we don't assume this in our optimization model, i.e. there is no restriction on the relation between s_f and p_f.

A machine commonly hosts many environments, thus supporting parallel execution of tasks. Our machine corresponds to a single OpenWhisk worker node, thus it may be a VM running on a IaaS cloud or a bare-metal node. Since the moment the environment's preparation starts — and until it is removed — each environment E_f uses q_f of the machine's resources (e.g., bytes of memory) whether a task executes or not. The number of simultaneously hosted environments is limited by the capacity of the machine Q (e.g. total amount of available memory; $\sum q_f \leq Q$). We consider only a single dimension of the resource requests as OpenWhisk, as well as Google Cloud Functions and AWS Lambda, allow customers to specify only the memory requirement — the amount of CPU power is determined by memory limit. However, it should be relatively easy to extend our model to (multi-dimensional) vector packing [9].

We do not consider the cost of the communication between tasks as the dependent functions exchange negligible amount of data, compared to a high-throughput, low-latency network of a modern datacenter. We assume that the machines are homogeneous (machine resources Q and execution times p_f are the same). If a FaaS system is deployed on VMs rented from an IaaS cloud, it is natural to use a Managed Instance Group (MIG) that requires all VMs to have the same instance type. If FaaS is deployed on a bare-metal data-center, the amount of machines having the same hardware configuration should be higher that other scalability limits (e.g. in a Google data-center, 98% of machines from a 10,000-machine cluster belong to one of just 4 hardware configurations [33]).

We assume that *jobs* have no release times, i.e., the first tasks of all the jobs are ready to be scheduled at time 0. This assumption approximates a system under peak load, when we observe temporary, rapid growth of incoming requests — there is a queue of pending requests to be scheduled at approximately the same time. Note that in contrast to *jobs*, individual tasks (in particular, the tasks that follow the first task of a job) do have non-zero release times, resulting from inter-task dependencies.

Our model is *clairvoyant*. A FaaS system repeatedly (thousands of times) executes individual functions. Thus, once a particular family is known for some time, q_f, p_f and the function structure should be easy to estimate using standard statistical methods — and before that, the system can use conservative upper bounds (e.g., defaults used by OpenWhisk). [23] shows that even simple methods estimate precisely memory and CPU requirements for long-running containers, which, in principle, is harder than estimating for FaaS systems, as functions in FaaS systems are shorter, thus repeated much more frequently.

The system optimizes the average response latency. As all N jobs are ready at time 0, this metric corresponds to $\frac{1}{N}\sum_{i=1}^{N} C_i$.

To summarize, the scheduling problem consists of finding for each task $O_{i,k}$ a machine and a start time $\sigma_{i,k}$ so that:

1. at $\sigma_{i,k}$, there is a prepared environment for $f(O_{i,k})$ on that machine that does not execute any other task during $[\sigma_{i,k}, \sigma_{i,k} + p_f)$ (a scheduling constraint);
2. $O_{i,k}$ starts after all its predecessors complete: $\sigma_{i,k} \geq C_{i,j}, \forall j \in P_{i,k}$ (a dependency constraint);
3. at any time, for each machine, the sum of requirements of the installed environments is smaller than the machine capacity (a knapsack-like constraint).

Algorithm 1 Framework scheduling algorithm.

```
function SCHEDULINGSTEP(t, queue, wait, policy)
                          ▷ policy ∈ {default, start}, wait ∈ {true, false}
  for task ∈ FINISHEDTASKS(t) do
    if policy == default then
      QUEUEDEPENDENTTASKS(task, t)
  for task ∈ ORDER(queue) do
    e ← FINDUNUSEDENVIRONMENT(task)
    if e is nil and wait then
      e ← FINDENVIRONMENTTOWAIT(task)
    if e is nil then
      e ← PLACENEWENVIRONMENT(task)
    if e is nil then
      e ← REMOVEANDPLACEENVIRONMENT(task)
    if e is not nil then
      ASSIGNTASK(c, task, RELEASETIME(task))
      REMOVEFROMQUEUE(task)
      if policy == start then
        p ← DURATION(task)
        QUEUEDEPENDENTTASKS(task, t + p)
```

This problem is NP-hard, as generalizing several NP-hard problems (bin-packing [14], $P2|DAG|\sum C_i$ [6]). A bin-packing instance can be encoded as an instance of our problem with items to pack corresponding to 1-task jobs (each from a distinct family, and the task size q_f equal to the size of the item to pack). With all the processing times $p_f = 1$, if the instance can be scheduled on m machines so that all tasks finish at time 1, this corresponds to packing items on m bins. Similarly $P2|DAG|\sum C_i$ can be encoded by setting q_f all to 1; capacity of both ($m = 2$) machines to 1 ($Q = q_f$) (so that a machine always executes at most one task); and having each tasks in a separate family, with processing times p_f equal to processing times of tasks in the $P2|DAG|\sum C_i$ instance.

3. Algorithms

In this section we describe heuristics to schedule FaaS invocations. We decompose the FaaS scheduling problem into three aspects: *sequencing* of invocations; *deployment* of execution environments on machines; and *allocation* of invocations to deployed environments. We start with a framework algorithm (Algorithm 1) to show how these aspects are combined to build a schedule; we then describe for each of the aspects several specific heuristics. *Sequencing* corresponds to the ordering policy (Section 3.1) and the awareness of task dependencies (Section 3.4). *Deployment* corresponds to the removal policy (Section 3.2). *Allocation* corresponds to the waiting/non-waiting variants (Section 3.3).

The framework algorithm is a standard scheduling loop executing *schedulingStep* at time t when at least one task completes. The algorithm maintains a queue of tasks $[O_{i,k}]$ to schedule and proceeds as follows:

1. Queue the successors of tasks completed at t ($\{O_{i,k} : \sigma_{i,k} + p_f = t\}$) if all their dependencies are already scheduled, along with their *release time* (*queueDependentTasks*). We maintain information about task's *release time* during scheduling process to ensure that dependency constraints are met (in particular the task may wait for completion of its dependencies after being assigned to the environment — we describe this case later).
2. Apply a scheduling policy to the queued tasks (*Order*). We describe policies for this step in Section 3.1.

3. Try to find an environment e for each queued task. Our goal is to avoid unnecessary setup of environments, therefore we take the following steps:
   (a) Try to claim an initialized environment of the required type (*FindUnusedEnvironment*, and — if *wait* — *FindEnvironmentToWait*). In this step we iterate over all machines and take the first matching unused environment. Section 3.3 describes action taken in the *wait* variant.
   (b) If (a) fails, try to create a new environment without removing any existing one (*PlaceNewEnvironment*). As above, we use the first machine that fits.
   (c) If (b) fails, try to find a machine with sufficient capacity for e that is currently claimed by environments that do not execute any task; remove these idle environments, and install e (*RemoveAndPlaceEnvironment*).
   (d) If (c) fails, the task remains in the queue.
4. If an environment e is found, assign the task (*AssignTask*); otherwise (3.a-c all fail) the task remains in the queue.

After each iteration of the main loop, the time t is shifted to the lowest completion time of the running tasks (in an implementation in a runtime system, the loop would block until the next task completes). *AssignTask* starts a task on an environment as follows. Each environment has a queue of assigned tasks. Immediately after creating an environment, it is initialized (which takes time s_f). Then, the environment starts to execute tasks sequentially from its queue. If the head task is not ready (waiting for dependencies), the environment waits (no backfilling). This may happen in the *start* policy (see Section 3.4).

In the following, we propose concrete variants for these functions. We denote the full scheduling policy by a tuple (A, B, C, D) where A denotes the tasks' ordering policy, B denotes the environments' removal policy, C indicates if variant is *waiting* and D describes whether the variant is dependency-aware, e.g., $(FIFO, LRU, wait, start)$.

3.1. Ordering policy (Order)

We compare the baseline FIFO and SJF policies with four policies taking into account the compositions and setup times:

- *FIFO (First Come First Served)*: use the order in which the tasks were added.
- *SJF (Shortest Jobs First)*: order by increasing tasks' durations p_f;
- *EF (Existing First)*: partition the tasks into two groups: (1) there is at least one idle, initialized environment e of matching type $E_{f(O_{i,k})}$; (2) the rest. Schedule the first group before the second group. The relative order of the tasks in both groups remains stable (FIFO). For example, if queue contains five tasks $[O_{i_1,k_1}, O_{i_2,k_2}, O_{i_3,k_3}, O_{i_4,k_4}, O_{i_5,k_5}]$, there is only one environment e that is idle and only tasks O_{i_1,k_1}, O_{i_3,k_3}, O_{i_4,k_4} require environment with type matching e, the resulting order is $[O_{i_1,k_1}, O_{i_3,k_3}, O_{i_4,k_4}, O_{i_2,k_2}, O_{i_5,k_5}]$.
- *SW (Smallest Work)*: order by increasing remaining sum of work of the task and its successors. This extends the SJF principle by taking into account the whole remaining work to be processed for the job, rather than just the ready task.
- *SP (Smallest Work on Critical Path)*: order by increasing remaining sum of work of the task and its successors on critical path (the longest path between the current state of each job and its completion). This extends the SW principle by taking into account the structure of the job: jobs with higher degree of parallelism will be favored.
- *RT (Release Time)*: order by the time the task's predecessors are completed.

3.2. Removal policy

When choosing an idle environment to remove, *RemoveAndPlaceEnvironment* removes environments according to either a baseline LRU policy, or one of policies considering either initialization time s_f or environment popularity:

- LRU: remove the least recently used (LRU) environment(s) from the first fitting machine (i.e. having enough space to be freed).
- min time removal: remove the environment(s) with the smallest setup time s_f (if more than one, select a single machine having environments with the smallest total s_f).
- min family removal: remove the idle environment(s) from the family with the highest number of currently initialized environments. As it may be needed to remove more than one environment, choose a machine to minimize the resulting number of families without any environment.

3.3. Greedy environment creation

If there is no unused environment of the required type E_f, a greedy algorithm just attempts to create a new one. However, when setup times s_f are longer than task's duration p_f, it might be faster just to wait until one of currently initialized environments completes its assigned task. We implement this policy by setting *wait* to *true* in Algorithm 1. When no idle environment is available, function *FindEnvironmentToWait* computes for each initialized environment e of type E_f the time C_e the last task currently assigned to this environment completes. If an environment e^* is available sooner than the time needed to set up a new environment ($\min C_e \leq t + s_f$), the task is assigned to e^*. This variant use the (limited) clairvoyance of the scheduler by taking into account the knowledge of tasks' durations and setup times of their execution environments.

The *waiting* variant is analogous to scheduling tasks in Heterogeneous Earliest Finish Time (HEFT [5,39]) that places a task on a processor that will finish the task as the earliest.

3.4. Awareness of task dependencies

A myopic (*default*) scheduler queues just the tasks that are currently ready to execute: $O_{i,0}$ (the first tasks in the jobs), or the tasks for which the predecessors completed $\{O_{i,k} : \forall_{j \in P_{i,k}} C_{i,j} \leq t\}$. However, when a task's $O_{i,k}$ predecessors complete, it might happen that there is no idle environment $e_{f(O_{i,k})}$, and thus $O_{i,k}$ must still wait s_f until a new environment is initialized.

We propose two policies, *start* and *start with break (stbr)*, that use the structure of the job to prepare environments in advance. Assume $O_{i,l}$ is the currently-scheduled task. These policies queue successors of $O_{i,l}$ when all predecessors completion time can be estimated. Of course, these successors are not yet ready to be executed (as their predecessors have not yet completed). We thus introduce the notion of the release time for each successor. These release times can be easily computed: as for each task we know its processing time p_f, the release time for each of the task's successors can be computed by the maximum completion time among its predecessors.

Note that *start* and *stbr* may result in an environment that is (temporarily) blocked: e.g., if an empty system schedules a chain of two tasks, the second task from the chain is added to the queue immediately after scheduling the first task; this second task will be assigned to its environment, but cannot be started until the first task is completed. In *start* variant, after *schedulingStep* completes and new tasks were added to queue, scheduler tries placing

them following the same procedure. Compared with *start*, *stbr* immediately after adding $O_{i,k}$ successor reorders tasks in the queue according to the scheduling policy and restarts the placement (for clarity, *stbr* is not presented in Algorithm 1).

4. Evaluation

We evaluate our algorithms with a calibrated simulator. We use a simulator rather than modify the OpenWhisk scheduler for the following reasons. First, a discrete-time simulator enables us to execute much more test scenarios and on a considerably larger scale (we simulate 1440 test instances each on 15 machine environments; Section 4.1 gives details on how we generate them). Second, as our results will show, to schedule tasks more efficiently, the OpenWhisk controller (the central scheduler) should take over some of the decisions currently made by the invokers (agents residing on worker machines). For example, *min family removal* needs to know which family has the highest number of installed environments in the whole cluster — thus, the state of the whole cluster (note that this policy can be implemented in a distributed way: the cluster state can be broadcasted to the invokers). To ensure that our simulator's results can be generalized to an OpenWhisk installation, in Section 4.2 we compare the performance of an actual OpenWhisk system with its simulation. We observe high Pearson correlation coefficient and a high coefficient of determination, confirming the realism of our simulation.

4.1. Method

We tested the performance of our algorithms on two kinds of test instances. First, we use synthetic test instances with a wide range of parameter values to test the general trends. Second, in Section 4.10, we adopt a recently-published Microsoft Azure Trace [26] to our model: there, we generate randomly only the data missing in the trace (such as the setup times).

Many parameters of test instances have a relative, rather than absolute, effect on the result. For example, multiplying by a constant both Q, the machine capacity, and q_f, the size of the task, results in an test instance that has very similar scheduling properties. There is a similar relationship between setup times s_f and durations p_f; and between the total number of tasks n and the number of tasks in a job l. We thus fix one parameter from each pair to a constant (or a small range); and vary the other.

In each simulation we use m machines of capacity Q. We have $n = 1000$ tasks assigned to n_f families. p_f is generated by the uniform distribution over integers $p_f \sim U[1, 10]$; similarly $q_f \sim U[1, 10]$. The remaining parameters have ranges:

- family count n_f: 10, 20, 50, 100, 200, 500;
- setup times s_f: $[0, 0]$, $[10, 20]$, $[100, 200]$, $[1000, 2000]$;
- number of tasks in a job (size of a job) l: $[2, 10]$, $[10, 20]$, $[50, 100]$;
- machine count m: 2, 5, 10, 20, 50;
- machine capacities Q: 10, 20, 50.

For each combination of the parameters (or ranges) n_f, s_f, l, we generate 20 random test instances, resulting in 1440 test instances. We evaluate each test instance on each of the 15 possible machine environments.

These ranges of parameters are wide. As we experiment on synthetic data, one of our goals is to explore trends — characterize test instances for which our proposed method works better (or worse) than the current baseline. In particular, jobs larger than 10 ($l > 10$) may have longer critical path than what we suspect is the current FaaS usage. On the other hand, it is not a lot compared with a call graph depth in any non-trivial software. At this point of FaaS evolution it is difficult to foresee the degree of compartmentalization future FaaS software will have — and DAGs larger than 10 invocations represent fine-grained decomposition (similar to the modern non-FaaS software).

We consider two sets of test instances: DAGs and chains. While DAGs fully express function compositions, we consider chains as an important case as they are directly supported by OpenWhisk platform — therefore results of our research can be applied to a real system. Moreover, chains enable us to validate our simulator against OpenWhisk (Section 4.2).

In FaaS system a single function is able to spawn an arbitrary number of other functions by connecting directly to the platform API. In general, executing DAGs by appending to each function code invoking successors using the platform's API hides the structure of the DAG from the scheduler. While spawning a new function using the API is straightforward, defining function that has more than one predecessor without direct platform support is more sophisticated, as it requires e.g. to store information about which of the predecessors completed their execution. In our analysis we assume that scheduler has information about defined DAGs and we analyze platform supporting function compositions that all forms of DAGs.

We generate a chain test instance as follows. Given n_f, $[s_{\min}, s_{\max}]$, $[l_{\min}, l_{\max}]$, for each of n_f, we set $s_f \sim U[s_{\min}, s_{\max}]$ and $p_f \sim U[1, 10]$. For each of $n = 1000$ tasks, we set its family f to $U[1, n_f]$. We then chain tasks to jobs. Until all tasks are assigned, we are creating jobs by, first, setting the number of tasks in a job to $l \sim U[l_{\min}, l_{\max}]$ (the last created job could be smaller, taking the remaining tasks); and then choosing l unassigned tasks and putting them in a random sequence.

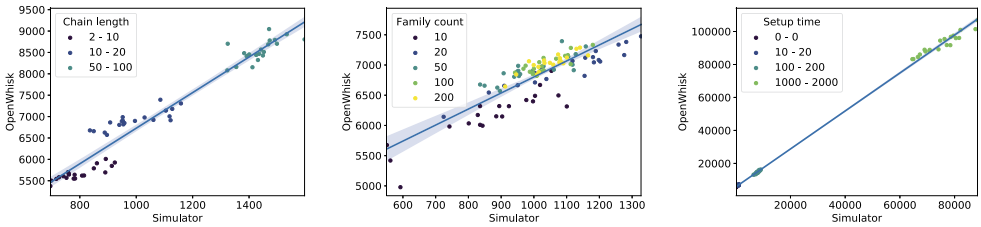
We generate DAGs similarly, but we change the algorithm to determine the dependencies. Given l tasks for a job, we first randomly permute them; then, for each k-th task in the permutation (except the first task), we generate the number of its predecessors χ from the uniform distribution, $\chi = |P_{i,k}| = U[1, k-1]$; and then select these χ predecessors as a random subset of size χ of the set $\{O_{i,1}, \ldots, O_{i,k-1}\}$.

For each experiment, our simulator computes the average response latency, $(1/n)\sum C_i$. Due to space constraints, we omit results on tail, 95%-ile latency — the 95%-ile results also support our conclusions (unsurprisingly, the ranges are larger than for the averages).

In addition to testing variants of Algorithm 1, we simulate the current, round-robin behavior of the OpenWhisk scheduler (Section 2.1) with an algorithm *OW*. *OW* randomly selects for each family f the initial machine m_f and the *step size* k_f, an integer coprime with the number of machines m. When scheduling a task $O_{i,k}$ in family f, *OW* checks machines m_f, $m_f + k_f$, $m_f + 2k_f$, ...(all additions modulo m), stopping at the first machine that has either the environment E_f ready to process, or q_f free resources (including unused environments that could be removed) to install a new environment E_f. If there is no such machine, $O_{i,k}$ is queued on a randomly-chosen machine.

4.2. Validation of the simulator against OpenWhisk

To compare the results of our simulator with OpenWhisk, we developed a customized OpenWhisk execution environment that emulates a function with a certain setup time s_f, execution time p_f and resource requirement q_f. We chose 10 ms as the time unit to reduce impact of possible fluctuations of VM or network parameters in the datacenter (we performed some early experiments with 1 ms and this noise was significant; and with a longer time unit tests take unreasonable time). This environment em-



(a) by chain length ($n_f = 50,\ s_f \in [10, 20]$)
(b) by family count ($l \in [10, 20],\ s_f \in [10, 20]$)
(c) by setup time ($n_f = 50,\ l \in [10, 20]$)

Fig. 2. Average latency on OpenWhisk system (Y axis) and simulated OW policy (X axis) with linear regression model fit. 1 unit is 10 ms. Each point corresponds to a single test instance executed on both OpenWhisk and simulator. Translucent bands indicate the 95% confidence interval.

Table 1
The 5th percentiles, medians and 95th percentiles of R^2 across obtained 1000 scores to verify the quality of the linear regression fit in Fig. 2.

Group	5th percentile	Median	95th percentile
Fig. 2(a)	0.86	0.93	0.97
Fig. 2(b)	0.29	0.67	0.84
Fig. 2(c)	0.997	0.998	0.9995

ulates initialization by sleeping for $s_f * 10$ ms; and it emulates execution by sleeping for $p_f * 10$ ms. While sleeping does not use the requested memory ($q_f * 128$ MB), the memory is blocked (through Linux cgroup limits) and therefore cannot be simultaneously used by other environments. We emulate a single test instance from our simulator by creating, for each job J_i, an equivalent sequence of invocations in OpenWhisk. To avoid caching of results in OpenWhisk, we ensure that each invocation is executed with a distinct set of parameters. We deployed an OpenWhisk cluster (1 controller and $m = 10$ invokers) on 11 VMs in Google Cloud Engine (GCE) in the *us-central-1a* zone. All machines have 2 vCPU and 16 GB RAM, and were running Ubuntu 18.04 LTS. We further restrict the memory OpenWhisk can use on machines to 1280 MB (equivalent to $Q = 10$). In order to reduce impact of cloud storage on system performance, we used a ramdisk to store OpenWhisk accounting database. We also extended limits (maximum duration and sequence length) and changed the default log level to WARN. To reduce the impact of brief performance changes, we executed each test instance thrice and reported the median.

In Fig. 2 we compare the average response latency in OpenWhisk and in our simulator varying chain lengths, the number of families and the ranges of setup times. For consistency, OpenWhisk results are rescaled to the simulator time unit (divided by 10). We use standard *Pearson correlation coefficient* [34] to validate correlation between results obtained from simulator and OpenWhisk. In particular, we compute the coefficient between X, the vector of average latencies as computed by our simulator for the OW policy, and Y, the vector of average latencies measured on OpenWhisk (a single element of these vectors corresponds with the measurement for a single instance). The Pearson correlation coefficient between OpenWhisk and simulator is very high (between 0.86 when varying family count, Fig. 2.b, and 0.999 when varying the setup time, Fig. 2.c). To further test our claim, we compute the coefficients of determination (R^2) scores [18] to verify the quality of the linear regression fit. We use the standard 5-fold cross-validation and repeat cross-validation 200 times (randomly permuting the data for each repetition). The 5th percentiles, medians and 95th percentiles of R^2 across obtained 1000 results are presented in Table 1. Thus, the R^2 scores are approximately equal to the squares of the Pearson correlation coefficients.

There is, however, an additive factor in OpenWhisk noticeable especially in smaller test instances in Fig. 2.(a) and Fig. 2.(b): the range of OpenWhisk results in $[5000, 9000]$, while the range of simulated results is in $[550, 1600]$; on larger test instances, as in Fig. 2.(c), this constant factor is less noticeable. This additive factor is caused by an additional system overhead added to every function execution: each invocation stores data in a database and requires internal communication. We conclude that the high correlation between the simulator and the OpenWhisk results validates our simulator — that the differences between algorithms observed in the simulator are transferable to the results in OpenWhisk. In the remaining sections we analyze results obtained from the simulator.

4.3. Relative performance of policies

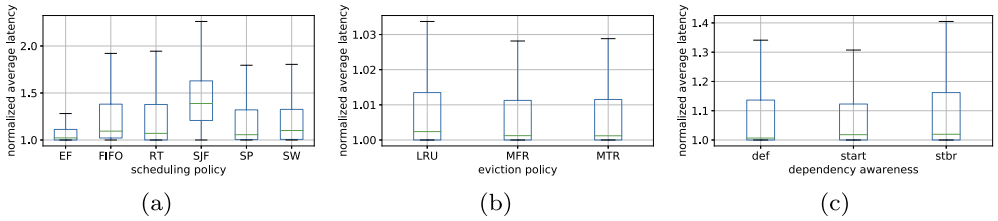
We first analyze the impact of each policy by analyzing their relative performance. For each variant (A, B, C, D), on each test instance, we compute the relative performance of the policy we measure by finding the minimal average latency across all variants of the measured policy while keeping the rest of the variants the same. For example, when measuring the effect of the scheduling policy (A), on an test instance, we find the minimum average latency from the 5 variants of the scheduling policy: (EF, b, c, d), (FIFO, b, c, d), (RT, b, c, d), (SJF, b, c, d), (SW, b, c, d), (SP, b, c, d) (keeping b, c, d the same); and then we divide all 5 by this value. The goal of this analysis is to narrow down our focus to the aspects of the problem that are crucial for the performance. Using this method, we show that, e.g., all removal policies result in very similar outcomes. Fig. 3 shows the results. Each box corresponds to a statistics over experiments with all the removal policies (both in *waiting* non-*waiting* variant) and all dependency-awareness variants (def, start, stbr), performed on all test instances and all possible machine environments (over 300k individual data points).

Ordering: *EF* policy dominates other ordering policies, confirming that it is better to avoid environment setup by reusing existing environments. Its median is similar to RT (and lower than other algorithms), and the range of values (including the third quartile) is the lowest.

Removal: Unlike scheduling policies, all the removal policies result in virtually the same schedule length: the range of Y axis is 1.035; thus outliers are only 3.5% worse than the minimal schedule found in the alternative methods.

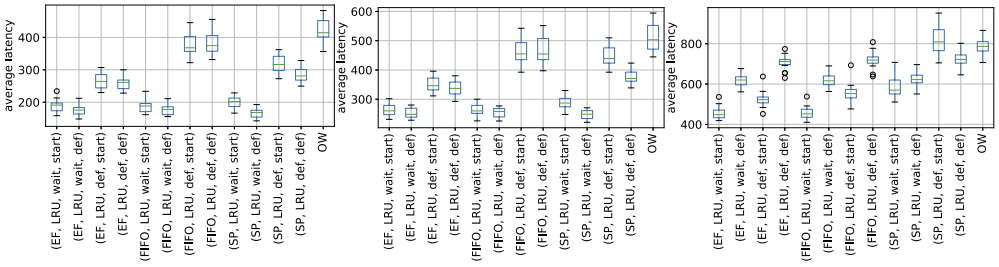
Dependency awareness: Both *start* and *stbr* result in similar performance. We confirmed this result by looking at individual test instances: the performance of *start* and *stbr* were similar.

To improve the readability in the remainder, given that the removal policies have little effect on the schedule length (Fig. 3), we show only the results for LRU. Similarly, we skip results for SJF and RT orderings: RT is close to FIFO and SJF is clearly dominated by other variants. SW and SP give similar results, thus we show



(a) (b) (c)

Fig. 3. Comparison of resulting average latency under: different scheduling policies (a), removal policies (b) and variants of dependency-awareness (c). For (b) and (c) results are normalized as in (a), but for different removal policies (b) and for different dependency-aware variants (c), rather than scheduling policies. Here and in all following box plots, the box height indicates the first and the third quartile, the line inside the box indicates the median, and the whiskers extend to the most extreme data point within 1.5 × IQR.



(a) length 2–10 (b) length 10–20 (c) length 50–100

Fig. 4. Influence of the number of tasks in a job. For all test instances $n_f = 50$, $m = 20$, $Q = 10$ with setup times 10–20.

only SP. Finally, as the difference between *start* and *stbr* variants is small, we show results only for *start*.

4.4. Impact of the number of tasks in a job

In the rest of the experimental section, we analyze the sensitivity of the policies to various parameters of the test instance, starting with the number of tasks in a job. While we explore wide range of parameters, presenting all resulting figures would be impractical. Our goal is to present trends, thus in the rest of the experimental section we present figures with representative case and conclusions from all the experiments.

In Fig. 4, in all test instances $n_f = 50$, $s_f \in [10, 20]$, $m = 20$, $Q = 10$ — we carried out experiments for all sets of parameters, but as the trends are similar, for practical reasons we show only results for these. All scheduling algorithms using EF as the ordering policy significantly reduce latency compared to the baseline OW (1.06–2.65x), with larger reductions for smaller jobs. The *start* dependency-aware variant further reduces latency, especially for jobs with more tasks ($[50 - 100]$), and also for other scheduling methods (FIFO). Therefore, for deployments with large (50 tasks and above) jobs, at least 100 families, setup times 100 (and larger) with at least 20 machines of capacity 10 (or more), implementing dependency-aware scheduler can provide measurable benefits.

4.5. Impact of the number of families

Fig. 5 compares results as a function of the number of task families in the system. When the number of task families is small (up to 20), variants without dependency awareness (*def*) and with *wait* can give better results than dependency-aware variants. In such cases, variants using EF method are slightly better than their equivalents using FIFO. The same applies to the removal method: *wait* variants give better results than their equivalents using plain LRU. The higher the number of families, the higher the probability that the required type of environment is missing. With at least $n_f = 100$ families (Fig. 5.c, similar results for $s_f \geq 100$, $l \geq 50$, $m \geq 20$, $Q \geq 10$ omitted to improve overall readability), dependency awareness plays a crucial role — variants using *start* outperforms *def* regardless of the used scheduling algorithm and removal policy. Thus, in case of high variability of functions (i.e. requiring different environments), taking into account tasks' dependencies can significantly reduce the serving latency.

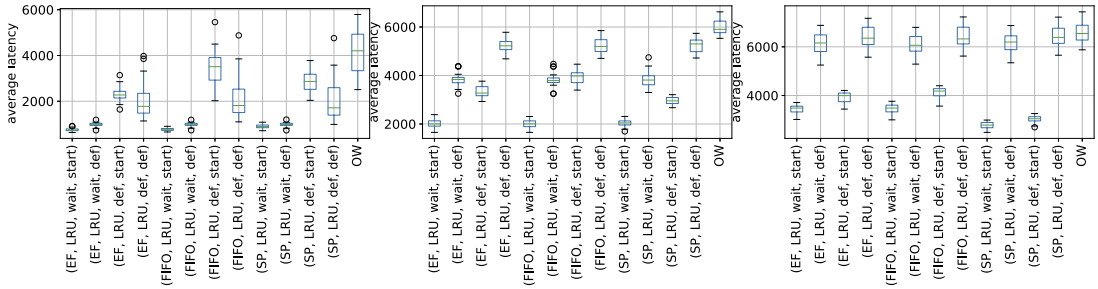
4.6. Impact of the setup time

Fig. 6 compares results as a function of different setup time ranges. In the edge case with no setup times, $s_f = 0$, we see no difference between the *waiting* and the non-*waiting* variants, as there is no additional penalty for inefficient environment recreation. Similarly, there are no differences between EF and FIFO. For non-zero setup times, dependency awareness (*start*) reduces the latency. However, with no setup time, *start* latencies are *longer*. This behavior is caused by adding tasks with future release time to the queue (see Section 3.4). Consider two jobs each of two tasks:

1. a *long* job with task *A* (duration 10) followed by task *B* (duration 1);
2. a *short* job with task *C* (duration 1), followed by task *B* (duration 1).

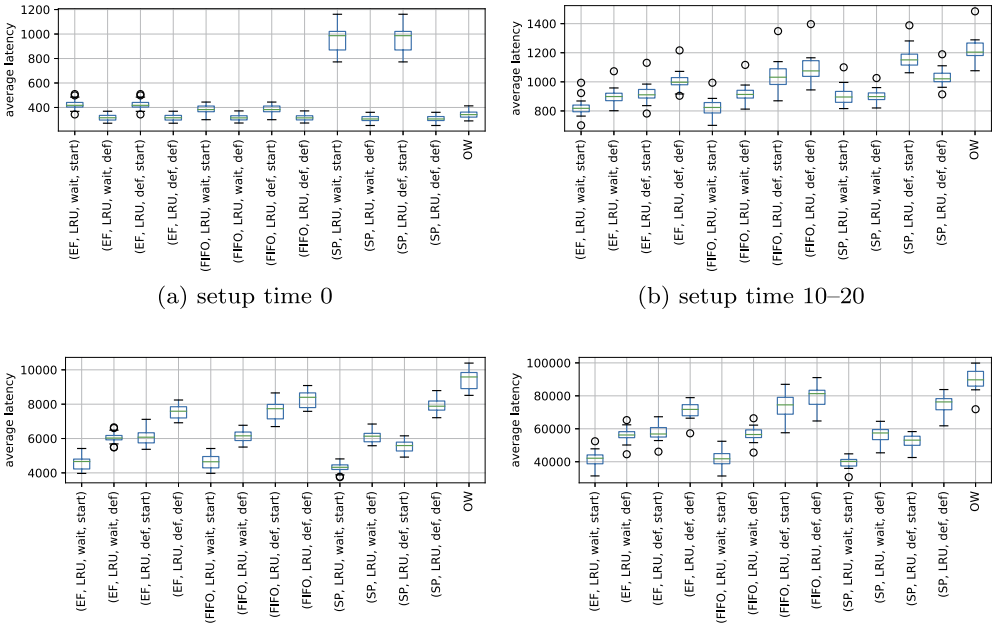
EF and *FIFO* using *start* variants may assign the second task from the *long* job to the environment of type B immediately after assigning the first task. This might block the second task from the *short* job until $t = 11$; while the optimal schedule starts this task at $t = 1$. For the same reason, *start* has worse results when there are more jobs (i.e. smaller jobs) and the systems are smaller (less machines, smaller capacities).

We further investigate for which test instance parameters the dependency-aware *start* dominates the myopic *def*, assuming non-negligible setup times $s_f \geq 100$. We aggregate results by all simulation parameters (count of families n_f, machines m, machine capacities Q, range of job sizes l, range of setup times s_f and used algorithm variant) and compute the median average latency among 20 test instances. Then we analyze in how many of resulting cases changing *def* to *start* improves performance. For large



(a) 10 families (b) 50 families (c) 200 families

Fig. 5. Influence of the different number of families. To show general trend, we present results for 10, 50 and 200 families. For all test instances $m = 20$, $Q = 10$, $s_f \in [100, 200]$, $l \in [50, 100]$.



(a) setup time 0 (b) setup time 10–20

(c) setup time 100–200 (d) setup time 1000–2000

Fig. 6. Influence of the setup time. For all test instances $n_f = 50$, $m = 10$, $Q = 10$, $l \in [50, 100]$.

jobs ($l \geq 50$), many task families ($n_F > 100$), and many machines ($m \geq 10$), changing the default (*def*) variant to dependency-aware one improves performance in all cases.

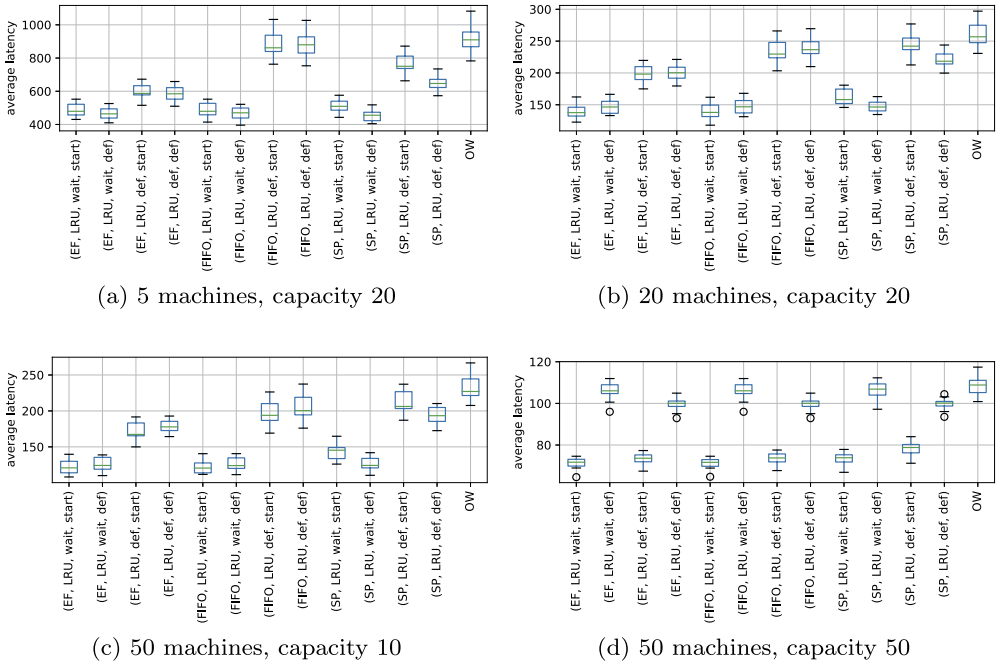
4.7. Impact of machine capacity

Fig. 7 compares results as a function of the number of machines and their capacity. For all test instances $n_f = 50$, $l \in [10, 20]$, $s_f \in [10, 20]$. To show general trend and ensure clarity, out of 15 considered machine configurations we present results only for test instances with $(m, Q) \in \{(5, 20), (20, 20), (50, 10), (50, 50)\}$. For cases up to $(m, Q) = (5, 20)$, the only non-negligible differences between the plain and dependency-aware variants are for SW and SP scheduling policies. Due to large number of jobs (job sizes are in range 10–20), when dependent tasks are added to the queue earlier, environments may get blocked as described in Section 4.6, therefore there is no additional benefit of dependency-awareness. For capacities up to $(m, Q) = (50, 10)$, using *wait* variants outperform the default (def) variants using the same scheduling algorithm and with the same setting of dependency-awareness. In all presented cases, for *FIFO* and *EF* scheduling policies, variants using *wait* with *start* have one of the lowest average latency. The improvement on overall system performance is most visible in the case of highly-overloaded machines. Therefore, our methods could be used to improve handling of situation when datacenter has to handle rapid increase (peak) of requests.

4.8. Summary of experiments

We conclude the experiments over DAGs by presenting averages improvements of $(\cdot, \cdot, wait, start)$ variants over OW baselines (Table 2). For each test instance on each particular machine configuration, we simulate each variant $(\cdot, \cdot, wait, start)$, compute the resulting average latency $\sum C_i$; and then divide it by the latency of the OW baseline. We then average these relative improvements across all variants and all test instances (including all possible machine configurations): this gives us the average impact of $(\cdot, \cdot, wait, start)$, regardless of the sequencing method. Thus, a number in Table 2 is an average over 12960 simulations: 3 (number of tasks in a job) times 6 (family count) times 2 (starting times settings) times 20 (repetitions) times 6 (ordering policies) times 3 (removal policies).

For non-negligible setup times (i.e. at least 100) in all machine configurations when the scheduler is dependency- and startup-times aware $(\cdot, \cdot, wait, start)$ the average response latencies are reduced at least by the factor of two.



(a) 5 machines, capacity 20

(b) 20 machines, capacity 20

(c) 50 machines, capacity 10

(d) 50 machines, capacity 50

Fig. 7. Influence of the machine environment. For all test instances $f_n = 50$, $l \in [10, 20]$, $s_f \in [10, 20]$.

Table 2
Average relative improvement in the average response latency of $(\cdot, \cdot, wait, start)$ over the OW baseline.

m	Q		
	10	20	50
2	2.96	2.92	3.89
5	3.46	4.10	3.73
10	4.44	4.12	3.52
20	4.41	3.89	3.18
50	4.15	3.61	3.12

(a) $s_f \geq 100$

m	Q		
	10	20	50
2	1.40	1.30	1.23
5	1.33	1.23	1.24
10	1.27	1.22	1.26
20	1.26	1.28	1.30
50	1.33	1.35	1.36

(b) $s_f < 100$

4.9. Differences between DAGs and chains

In [40] we performed analysis analogous to Section 4.6, Section 4.4 and Section 4.7, but for chains, instead of DAGs.

In general, obtained results are similar: if we compare behavior for different setup times (Fig. 5 in [40] and Fig. 6), we can observe that for non-zero setup times our algorithms perform better than baseline (OW). In both DAGs (Fig. 7) and chains (Fig. 6 in [40]) only for the largest processing capacity (50 machines of capacity 50) there is observable fundamental improvement of dependency-aware (*start*) variants over *def*. Moreover, for all machine configurations except the largest one (50 machines of capacity 50), *wait* variants performed better than the default (*non-waiting*) variants using the same scheduling method and with the same setting of dependency-awareness.

Nevertheless, there are observable differences. Increasing l, the number of tasks in a job, increases average latency more significantly for chains (Fig. 3 in [40]) than for DAGs (Fig. 4). For the same number of tasks in a job, a chain has less available concurrency than a DAG (i.e. longer critical path). This also impacts differences observed in comparison of dataset with different setup times (and constant job size) — for chains (Fig. 5 in [40]) we observe higher difference between variants without (*def*) and with dependency awareness (*start*) than for DAGs (Fig. 9). For similar reasons, for datasets with more families, our algorithms for DAGs (Fig. 5) — as well as for chains (Fig. 4 in [40]) — provide lower average latencies than OW, but difference between the best schedule and the baseline for DAGs is noticeably smaller.

Overall, the results are similar, proving the robustness of our proposed algorithms. In all analyzed cases with non-zero setup times, our proposed algorithms — the *EF* scheduling policy with *LRU* replacement and *waiting* environment creation — outperform the OW baseline.

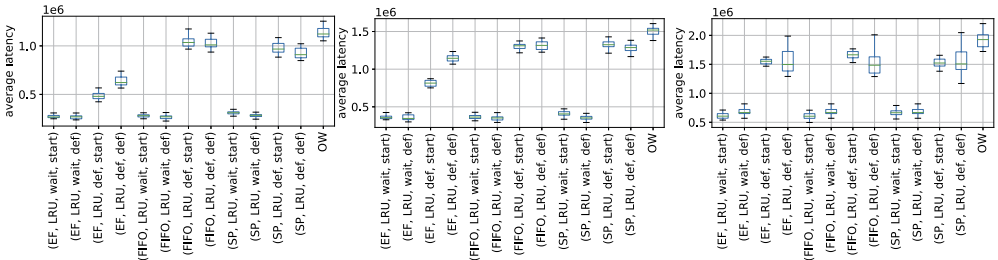
4.10. Experiments with a FaaS trace

To the best of our knowledge, there is no publicly available FaaS trace containing information about tasks with dependencies, setup times and tasks families (types or applications). However, based on an existing trace, we can generate a dataset making some rational assumptions about missing data. Thus, we can verify how such dataset-like workload would behave if executed in the analyzed model.

The recently-published Microsoft Azure Trace [26] contains information about sizes, durations and invocations patterns in Azure Functions. Memory usage is provided only for first 12 days of the trace, thus we limit our analysis to this range (as we will use memory data to generate q_f, the size of the environment).

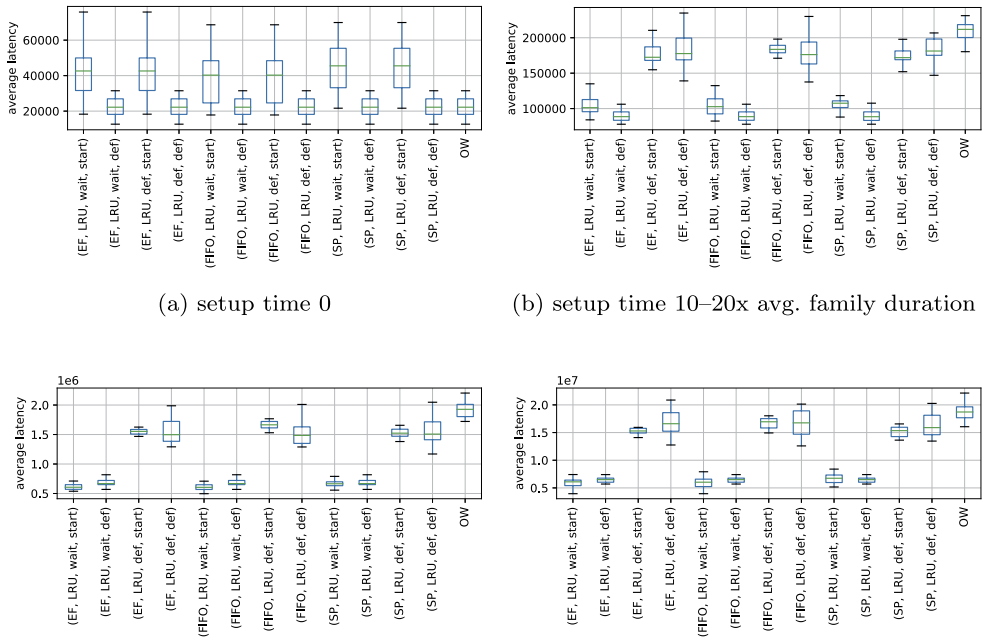
Functions in trace are organized in groups called *apps* which share common execution environment. As our model requires separate environments per function, we further narrow down our analysis to *apps* containing only one function. We map each *app* to a function family.

We set the environment size q_f to the maximum allocated memory (*100th percentile of the average allocated memory* column of Azure trace) as the environment should be large enough to han-



(a) 2–10 tasks (b) 10–20 tasks (c) 50–100 tasks

Fig. 8. Comparison of different job sizes. In each case dataset contains 50 families generated from Azure trace. Setup times 100–200 times larger than average duration observed over all families, 10 machines of capacity 20.



(a) setup time 0 (b) setup time 10–20x avg. family duration

(c) setup time 100–200x avg. family duration (d) setup time 1000–2000x avg. family duration

Fig. 9. Comparison of impact of setup times. In each case, setup times are obtained by multiplying average duration across all families by random value from given range. All datasets contains 50 families and 1000 tasks in jobs of 50–100 tasks generated from Azure trace. Experiments were run on 10 machines of capacity 20.

dle all the invocations. We then normalize q_f to range $\{1, \ldots, 10\}$ to have similar range of values as our synthetic test instances. We set the execution time p_f to the average execution time (*Average* column in the trace).

Similarly to synthetic datasets described in Section 4.1, we generate datasets for a wide range of remaining parameters: the number of families in the system n_f and setup times s_f. Setup time of a family is, in principle, independent of this family's execution time; yet, we need to control the relation of the mean setup time to the mean execution time (this relation is a parameter of our experiment). We thus compute the mean execution time $\overline{p_f}$ across n_f families in the test instance; and then generate setup times s_f from $\overline{p_f}\, U[S_{\min}, S_{\max}]$.

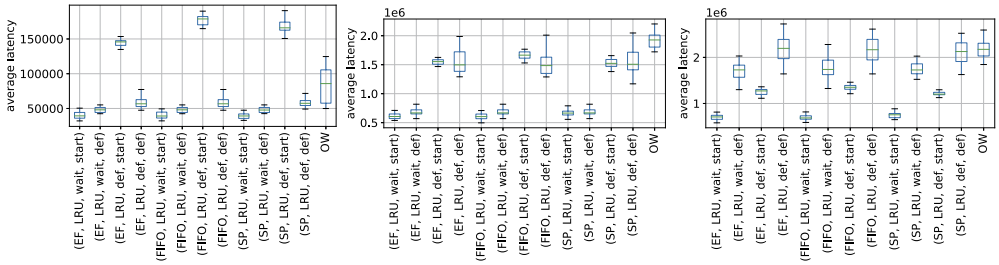
For each configuration, we generate a test instance containing $n = 1000$ tasks. For each test instance, we select n_f families generated from *app* data which have the largest invocation count within the considered 12-day period. We use information about *app* invocation counts to reflect invocation pattern within our sample: for each invocation we select its family randomly, but with weights proportional to the total number of invocation in the original trace. Then we generate DAGs (jobs) following the same procedure as in Section 4.1.

Fig. 8 shows the impact of different job sizes. All datasets have 50 families and setup times are 100–200 times larger than average family duration. In all cases enabling *waiting* improves performance. However for small jobs (lower than 50 tasks), the *start* variant gives no additional benefit.

Fig. 9 shows the impact of different setup times. Similarly to synthetic test instances, for configurations with negligible setup time, enabling dependency-awareness decreases performance. With considerable setup times (at least 100 times larger than the average duration), EF and FIFO with *waiting* and dependency-awareness (*start*) have one of the lowest average latencies.

In Fig. 10 shows the impact of the number of families. Variants with *waiting* and enabled dependency-awareness give better results than their non-waiting or not dependency aware equivalents. Moreover, when there are many families (200), dependency-awareness plays a crucial role — *start* variants overtake their *def* equivalents.

While generated samples have higher value of average latency than our synthetic datasets (as we don't normalize durations), we observe in all cases that enabling *waiting* reduces the latency. Analogously to results obtained for synthetic datasets, for non-negligible setup times, *FIFO* and *EF* variants with *waiting* and



(a) 10 families (b) 50 families (c) 200 families

Fig. 10. Comparison of different count of families generated from Azure trace. In all cases setup times are in range 100–200x average duration across all families. We present results for 10 machines of capacity 20, jobs containing 50–100 tasks (note: except last generated one).

enable dependency-awareness (*start*) give significant improvement over baseline (*OW*).

5. Related work

Our model of FaaS resource management combines scheduling (with setup times and dependencies) [2] with bin packing (when environments of different sizes must fit into machines). Our simulation results show that all these aspects have to be taken into account by the scheduler (the baseline *OW* is consistently dominated by our policies). Individually, these are classic problems in combinatorial optimization. Allahverdi [2] performs a comprehensive review of about 500 papers on scheduling with setup times. Brucker [6] reviews scheduling results. We start by describing the closest related combinatorial optimization approaches (these approaches are mostly theoretical or based on simulation). We then follow by a discussion on other systems-based approaches to optimization in FaaS.

Quadratic programming: We proposed heuristics, rather than generic solvers or metaheuristics. Initially, we considered encoding our problem as an (integer) quadratic programming. Nevertheless, Gurobi [16] was unable to find an optimal schedule in 15 minutes (on a reasonable desktop machine) even for a small test instance with $N = 20$ jobs each of $n_l = 20$ tasks. Schedulers in production systems need to respond in seconds, thus an approach based on a generic solver is probably not sufficient.

Bin packing with setup times: With no dependencies, our problem reduces to bin packing with sequence independent setup times. Weng et al. [36] study similar problem of minimizing mean weighted completion time for tasks with sequence dependent setup times. [35] presents dynamic algorithms addressing scheduling with setup times with objective of minimal weighted flow time.

Workflow scheduling: With no setup times ($s_f = 0$) and task sizes equal to machine capacities $q_f = Q$, our problem reduces to workflow scheduling. [37] surveys workflow scheduling in the cloud. [19] measures how inaccurate runtime estimates influence the schedules which complements our study, as we assumed that estimates are known. [27] analyzes possible performance benefits of resource interleaving across the parallel stages. [24] proposes Balanced Minimum Completion Time, an algorithm for scheduling tasks with dependencies (and without setup times) on heterogeneous systems. [13] schedules workflows with setup times using branch-and-bound. The evaluation in that paper considered small instances (up to $N * n_l = 100$ task and $m = 4$ machines); their method required 100 s time limit for execution. Such long running times makes this method unusable in data-center schedulers. [1] analyzes scheduling tasks with sequence-dependent setup times, precedence constraints, release dates on unrelated machines with resource constraints and machine eligibility. Two algorithms are analyzed: based on genetic algorithm and based on an artificial immune system. Their largest instances had 60 tasks and 8 machines and needed 25 minutes (on the average) to solve, again rendering these methods unusable for FaaS.

Systems approaches in Serverless Computing: Serverless is currently rapidly evolving with multiple ongoing efforts to analyze and extend it. In our work we explore possible performance improvements by considering the composition of functions. [7,15] also consider composed functions. [15] analyzes function chains and attempts to reduce the number of used containers while keeping response time below a predefined limit. Their implementations use Brigade running on top of Kubernetes. [7] analyzes function DAGs and possible benefits of storing intermediate results inside executors.

Our simulation results show that the scheduling matters especially when setup times of new environments are high. Reducing setup times has received considerable attention. [29] proposes checkpointing and then restoring environments. Catalyzer [10] reuses the environment state. Particle [32] identifies environment network configuration as an important contributor to the setup time in container-based platforms, including OpenWhisk; and proposes how to decouple that from environment creation.

Another key parameter is the size of the environment (q_f): the smaller the environments are, the more can run currently on a machine, thus packing and evictions become less crucial. The following approaches reduce the memory footprint of environments. [28] proposes new isolation abstraction reducing memory requirements. Photons [11] reduces overall memory consumption by running multiple concurrent invocations within a single environment (without impacting reliability). ENSURE [31] improves resource efficiency by adapting the resource usage of environments running on a single invoker and concentrating workloads to minimize number of concurrently running environments.

An orthogonal approach to reduction of the serving latency is to directly reduce p_f, the function's processing time. As in the classic FaaS model, the functions are stateless, getting the state from external storage takes time. Cloudburst [30] demonstrates and analyzes usability of stateful serverless computing. [38] presents framework for building stateful and fault-tolerant serverless-based applications, running on top of existing platforms. [22] shows another, orthogonal approach: they change scheduling of invocations on an individual node.

6. Conclusions

In the FaaS model, the time needed to create an environment for an incoming function invocation in most cases cannot be neglected. We predict that the growing popularity of FaaS systems will result in more complex applications being created in this model. As we show in this paper, the cloud provider can significantly optimize FaaS performance knowing the structure of

the compositions used in the workload. Our framework algorithm could be implemented in FaaS systems, however some changes in the architecture might be required, e.g. the Apache OpenWhisk controller would need to assign invocations directly to containers running on invoker nodes and also to directly create and evict the environments.

In our experiments, we identified the three policies in our framework algorithm that lead to largest improvements. Taken together, they consider the structure of the functions and thus they can install environments in advance. The *start* variant adds successors of each task to the queue so that the scheduler then knows what environments should be prepared in advance. The *waiting* variant, rather than greedily creating an environment for each task, binds a task to the existing, currently busy environment if such environment will be available to process the task earlier than a newly-created one. Finally, the *EF* ordering prioritizes tasks that can be started using already prepared environments.

Our simulation results clearly show that the performance of FaaS can be improved by these methods. For non-negligible setup times (i.e. at least 100 or at least 20 times longer than the average task duration) in all machine configurations when the scheduler is dependency- and startup-times aware $(\cdot, \cdot, wait, start)$ the average improvement of the response latency is at least by the factor of two. We summarize the observed improvements in Table 2.

Thus, our results indicate that dependency- and startup-times aware scheduling is more efficient when the load of the system is high. Our methods can be used to mitigate the impact of the increased demand in the short term. If the demand increase is longer-term, the underlying infrastructure will be eventually scaled out by, e.g., adding new VMs. However, such scale-out takes considerably longer time (minutes); meanwhile, the load has to be handled.

Although our experiments were offline, the *waiting* variant and the *start* variant can be easily implemented in the existing FaaS schedulers (controllers). Our results show that *waiting* and *start* variants are beneficial even with the standard FIFO ordering. Changing the invocation order (as in SJF and EF variants) is less straightforward, as when new jobs arrive on-line, existing jobs might be starved: these policies would additionally need to consider fairness.

Finally, while FaaS is the main motivation of this work, these ideas can be applied also in other systems executing workflows on shared machines (a machine executing multiple tasks in parallel), such as Apache Beam.

CRediT authorship contribution statement

Pawel Zuk: Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Krzysztof Rzadca:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is supported by a Polish National Science Center grant Opus (UMO-2017/25/B/ST6/00116).

References

[1] M. Afzalirad, J. Rezaeian, Resource-constrained unrelated parallel machine scheduling problem with sequence dependent setup times, precedence constraints and machine eligibility restrictions, Comput. Ind. Eng. 98 (2016).
[2] A. Allahverdi, The third comprehensive survey on scheduling problems with setup times/costs, Eur. J. Oper. Res. 246 (2) (2015).
[3] Apache OpenWhisk, https://openwhisk.apache.org, 2020.
[4] I. Baldini, P. Cheng, S.J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu, The serverless trilemma: function composition for serverless computing, in: SIGPLAN, Proc., ACM, 2017.
[5] L.F. Bittencourt, R. Sakellariou, E.R. Madeira, DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm, in: PDP, Proc., IEEE, 2010.
[6] P. Brucker, Scheduling Algorithms, Springer, 2007.
[7] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, Y. Cheng, Wukong: a scalable and locality-enhanced framework for serverless parallel computing, in: SoCC, Proc., SoCC '20, ACM, 2020, pp. 1–15.
[8] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (12) (2019) 44–54.
[9] C. Chekuri, S. Khanna, On multi-dimensional packing problems, in: SODA, Proc., 1999.
[10] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, H. Chen, Catalyzer: sub-millisecond startup for serverless computing with initialization-less booting, in: ASPLOS, Proc., ASPLOS '20, ACM, 2020, pp. 467–481.
[11] V. Dukic, R. Bruno, A. Singla, G. Alonso, Photons: lambdas on a diet, in: SoCC, Proc., SoCC '20, ACM, 2020, pp. 45–59.
[12] G.C. Fox, V. Ishakian, V. Muthusamy, A. Slominski, Status of serverless computing and function-as-a-service (FaaS) in industry and research, preprint, arXiv:1708.08028.
[13] B. Gacias, C. Artigues, P. Lopez, Parallel machine scheduling with precedence constraints and setup times, Comput. Oper. Res. 37 (12) (2010).
[14] M.R. Garey, D.S. Johnson, Computers and Intractability, vol. 174, 1979.
[15] J.R. Gunasekaran, P. Thinakaran, N.C. Nachiappan, M.T. Kandemir, C.R. Das, Fifer: tackling resource underutilization in the serverless era, in: Middleware, Proc., Middleware '20, ACM, 2020, pp. 280–295.
[16] Gurobi optimizer reference manual, http://www.gurobi.com, 2019.
[17] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Serverless computation with openlambda, in: HotCloud, Proc., USENIX Association, 2016.
[18] A. Hughes, D. Grawoig, Statistics: A Foundation for Analysis, 1971.
[19] A. Ilyushkin, D. Epema, The impact of task runtime estimate accuracy on scheduling workloads of workflows, in: CCGRID, Proc., 2018.
[20] K. Kritikos, P. Skrzypek, A review of serverless frameworks, in: UCC, Proc., 2018, pp. 161–168.
[21] Kubernetes, https://kubernetes.io.
[22] B. Przybylski, P. Zuk, K. Rzadca, Data-driven scheduling in serverless computing to reduce response time, in: CCGrid, Proc., IEEE, 2021.
[23] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, et al., Autopilot: workload autoscaling at Google, in: Euro-Sys, Proc., 2020.
[24] R. Sakellariou, H. Zhao, A hybrid heuristic for dag scheduling on heterogeneous systems, in: IPDPS, Proc., IEEE, 2004.
[25] M. Shahrad, J. Balkind, D. Wentzlaff, Architectural implications of function-as-a-service computing, in: ISM, Proc., MICRO '52, ACM, 2019, pp. 1063–1075.
[26] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, R. Bianchini, Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider, preprint, arXiv:2003.03423.
[27] W. Shao, F. Xu, L. Chen, H. Zheng, F. Liu, Stage delay scheduling: speeding up dag-style data analytics jobs with resource interleaving, in: ICPP, Proc., 2019.
[28] S. Shillaker, P. Pietzuch, Faasm: lightweight isolation for efficient stateful serverless computing, in: USENIX ATC, Proc., USENIX Association, 2020, pp. 419–433.
[29] P. Silva, D. Fireman, T.E. Pereira, Prebaking functions to warm the serverless cold start, in: Middleware, Proc., Middleware '20, ACM, 2020, pp. 1–13.
[30] V. Sreekanti, C. Wu, X.C. Lin, J. Schleier-Smith, J.M. Faleiro, J.E. Gonzalez, J.M. Hellerstein, A. Tumanov, Cloudburst: stateful functions-as-a-service, preprint, arXiv:2001.04592.
[31] A. Suresh, G. Somashekar, A. Varadarajan, V.R. Kakarla, H. Upadhyay, A. Gandhi, Ensure: efficient scheduling and autonomous resource management in serverless environments, in: ACSOS, Proc., IEEE, 2020, pp. 1–10.
[32] S. Thomas, L. Ao, G.M. Voelker, G. Porter, Particle: ephemeral endpoints for serverless networking, in: SoCC, Proc., SoCC '20, ACM, 2020, pp. 16–29.
[33] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, in: EuroSys, Proc., ACM, 2015.
[34] L. Wasserman, All of Statistics: A Concise Course in Statistical Inference, vol. 26, Springer, 2004.
[35] S. Webster, M. Azizoglu, Dynamic programming algorithms for scheduling parallel machines with family setup times, Comput. Oper. Res. 28 (2) (2001).

[36] M.X. Weng, J. Lu, H. Ren, Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective, Int. J. Prod. Econ. 70 (3) (2001).
[37] F. Wu, Q. Wu, Y. Tan, Workflow scheduling in cloud: a survey, J. Supercomput. 71 (9) (2015).
[38] H. Zhang, A. Cardoza, P.B. Chen, S. Angel, V. Liu, Fault-tolerant and transactional stateful serverless workflows, in: OSDI, Proc., USENIX Association, 2020, pp. 1187–1204.
[39] H. Zhao, R. Sakellariou, An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm, in: H. Kosch, L. Böszörményi, H. Hellwagner (Eds.), Euro-Par, Proc., Springer, 2003, pp. 189–194.
[40] P. Zuk, K. Rzadca, Scheduling methods to reduce response latency of function as a service, in: SBAC-PAD, Proc., IEEE, 2020, pp. 132–140.

Pawel Zuk is a PhD student in Computer Science at the Institute of Informatics, University of Warsaw, under supervision of Krzysztof Rzadca.

He earned his undergraduate and Master's degree in Computer Science within Inter-faculty Studies in Mathematics and Natural Sciences at University of Warsaw.

His research interests lie in scheduling and cloud computing.

Krzysztof Rzadca is an associate professor in the Institute of Informatics, University of Warsaw, Poland and a data science lead at Google. He graduated with an MSc in software engineering from Warsaw University of Technology, Poland in 2004, and a PhD in computer science in 2008 jointly from Institut National Polytechnique de Grenoble (INPG), France and from Polish-Japanese Institute of Information Technology, Poland. He was French government fellowship recipient during his PhD studies. Between 2008 and 2010, he worked as a research fellow in Nanyang Technological University (NTU), Singapore. He was awarded grants from the Polish National Science Center, the Foundation for Polish Science and a faculty research award from Google. His research focuses on resource management and scheduling in large-scale distributed systems, such as supercomputers and clouds.