University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Paweł Żuk

Resource allocation methods for serverless cloud computing platforms

PhD dissertation

Supervisor:
dr hab. Krzysztof Rządca

Warsaw, January 2023

Abstract

Serverless computing changes the way we use, operate, and think about cloud computing. Instead of using conventional Virtual Machines or containers, with Function as a Service (FaaS), cloud customers define a *function*: essentially a code snippet. Then, a FaaS system executes the function in an isolated environment in response to incoming events. This paradigm frees the cloud customer from the responsibility of provisioning and maintaining the underlying software stack. At the same time, FaaS workload is less opaque for the cloud provider, compared to VMs or containers – the design of FaaS and the (relatively) small size of functions allow more accurate estimation of function resource usage or execution times. The provider may use this knowledge to improve provisioning execution environments or change the order in the schedule.

In this dissertation, we analyze three aspects of resource management in FaaS. First, we study function composition: a function call may form a chain or, in general, a DAG. Knowing the internal structure of composition, the cloud provider can schedule these future invocations, prepare required execution environments in advance and optimize resource usage.

Second, we analyze scheduling and load balancing from the cluster perspective. We introduce a model connecting these aspects, where the resource manager is responsible for both scaling the number of instances (e.g., function execution environments) and distributing the workload between them.

Third, we focus on scheduling performed on a single node of a cluster. We specify possible enhancements and propose heuristics that benefit from information gathered locally on a node: recorded call frequencies and durations of previous function calls.

Our results show that the characteristic properties of FaaS allow for significant improvements in resource utilization, and the proposed solutions will bring benefits to both the cloud provider and the customers.

Keywords: Function as a Service, scheduling, workflow, serverless, cloud

Streszczenie

Obliczenia bezserwerowe zmieniają sposób użytkowania, zarządzania, a nawet myślenia o przetwarzaniu w chmurze obliczeniowej. W odróżnieniu od klasycznych maszyn wirtualnych bądź kontenerów, *Funkcja jako serwis* (Function as a Service, FaaS) pozwala klientom chmury na zadeklarowanie *funkcji*: fragmentu kodu wykonywanego w izolowanym środowisku w odpowiedzi na przychodzące zdarzenia. Paradygmat ten uwalnia klienta od konieczności tworzenia i zarządzania środowiskami obliczeniowymi. Jednocześnie obliczenia w modelu FaaS są bardziej przejrzyste w porównaniu z maszynami wirtualnymi lub kontenerami – pozwalają więc na dokładniejsze oszacowanie czasu przetwarzania oraz zasobów niezbędnych do jej wykonania. Korzystając z tej wiedzy, dostawca chmury może usprawnić harmonogram tworzenia środowisk uruchomieniowych lub zmienić kolejność wywołań.

W rozprawie analizujemy trzy aspekty alokacji zasobów dla FaaS. Pierwszym zagadnieniem jest kompozycja funkcji: wywołania funkcji mogą kształtować się w łańcuch lub, w ogólnym przypadku, graf acykliczny (DAG). Znajomość struktury grafu wywołań pozwala dostawcy chmury szeregować przyszłe wywołania, przygotować wymagane środowiska uruchomieniowe z wyprzedzeniem i optymalizować zużycie zasobów.

Drugim aspektem są problemy szeregowania oraz równoważenia obciążenia z perspektywy klastra. Wprowadzamy model łączący oba te aspekty, w którym zarządca zasobów odpowiada zarówno za skalowanie liczby instancji (np. środowisk uruchomieniowych funkcji) oraz za rozkładanie przychodzącego obciążenia między utworzone instancje.

Trzecią analizowaną kwestią jest szeregowanie wywołań funkcji na poziomie węzła wykonawczego. Analizujemy możliwe usprawnienia oraz proponujemy heurystyki wykorzystujące informacje rejestrowane na poziomie węzła: odnotowaną częstotliwość wywołań funkcji oraz czasy działania poprzednich wywołań.

Uzyskane wyniki wskazują, iż charakterystyczne cechy FaaS pozwalają na wprowadzenie usprawnień w mechanizmach zarządzania zasobami, które przyniosą korzyści dostawcy chmury oraz użytkownikom.

Słowa kluczowe: Function as a Service, szeregowanie, workflow, przetwarzanie bezserwerowe, chmura obliczeniowa

Acknowledgements

First, I would like to express my gratitude to my advisor Krzysztof Rządca for his guidance during my PhD studies, patience and kindness. This work would not reach its final form without his priceless support. I would like to thank my co-author Bartłomiej Przybylski for fruitful research collaboration and for creating a friendly atmosphere. I also could not be more grateful to my family for their understanding and unconditional support over the years.

This research was financially supported by National Science Center in Poland via grant "Optimization methods for allocation of large-scale computational resources" (2017/25/B/ST6/00116, PI: Krzysztof Rządca) and Google via research award "Algorithms for co-allocating services in datacenters" (2014-R2-722, PI: Krzysztof Rządca).

Contents

Chapter 1

Introduction

1.1 FaaS: A high-level abstraction layer in the cloud computing stack

Nowadays, clouds and supercomputers have a wide range of applications and we often do not even realize that we are using the effects of their work. We use their help when there is a need to parallelize calculations or to process large amounts of data that cannot fit in the memory of a single computer or a smartphone. In many situations, it is necessary to use many machines – for instance, in large-scale numerical calculations, such as weather forecasts or models of physical phenomena. Another use of cloud computing is the need to ensure the availability of Internet services. In that case, the cloud customer may launch multiple application instances and divide incoming traffic between them. That occurs, for example, in online services like search engines or social networking sites.

Supercomputers and clouds are composed of thousands of *nodes* – for instance, a typical *cell* in the Google cloud includes 12 000 machines [1]. Another example is the *Okeanos* supercomputer at the Interdisciplinary Center for Mathematical and Computational Modeling of the University of Warsaw, built from 1084 nodes [2]. The cloud provider can optimize the use of available resources, as well as reduce the cost of constructing and operating a data center (which includes, among others, the cost of electricity, network connection, maintenance of the building or cooling) [3], by sharing the equipment between multiple customers. Cloud computing also brings numerous advantages for cloud customers compared to dedicated infrastructure. In particular, the cloud customer often does not know about the actual resource demand, or the requirements may change due to external factors. Thus, it can be challenging to estimate the required size of the on-premise infrastructure. In contrast, cloud customers can flexibly change the amount of allocated resources.

As cloud computing has gradually gained mainstream adoption, different usage models emerged over the years. With the evolution of support for virtualization in modern CPUs, the cloud providers started offering access to computing platforms in IaaS (*Infrastructure as a Service*) model. In this model, customers can run a dedicated virtual machine (*VM*) in the cloud environment. This approach however brings several disadvantages: running an application in a dedicated VM requires more resources, compared to running it natively on a host, and brings performance penalty [4].

These issues led to growth in the popularity of *containers* ("lightweight Virtual Machines") — a method of creating separate environments which share only the operating system kernel (and the underlying hardware). On a larger scale, container deployment and management can be automated by using one of the orchestration platforms like Kubernetes [5] or Docker Swarm [6]. Nonetheless, the cloud customer is responsible for the environment inside the container: installation of required packages, configuration, and selection of which program to execute at container creation.

PaaS (*Platform as a Service*) model enables cloud customers to deploy and manage their applications without the need to set up the underlying infrastructure on their own. While the cloud customers do not manage internal cloud components like Virtual Machines, databases, or storage, they may be able to change the configuration parameters of the application environment [7]. Internally PaaS offerings often use containers as a standard interface for providing applications. However, although PaaS hides the complexity of the configuration of the underlying infrastructure, the management of autoscaling is still a complex task [8].

The trend of shifting from monolith applications to microservices and the widespread adoption of containers lead to the increasing popularity of *serverless computing* paradigm [8]. Notably, this paradigm introduces FaaS (*Function as a Service*) computational model. In this model, cloud customers (developers) declare functions – relatively small, stateless code fragments executed in response to incoming events (e.g., HTTP requests). The cloud provider manages FaaS infrastructure – the cloud customers (developers) do not have to maintain execution environments and scale them. At the same time, the cloud operators gain a better view of the characteristics of the application (consisting of many small functions), which opens up new opportunities for optimizing resource allocation. For example, the cloud provider can measure processing time and use these measurements to estimate the duration of incoming calls. This knowledge can be used to change the order of execution of the calls and improve end-user experience. Moreover, as the lifespan of the execution environment is under cloud providers' control, all environments might be removed if a reduction of used memory is needed. However, a lack of an available execution environment may lead to degradation of QoS, as the end-user has to wait longer for a response due to the time required to create an environment.

FaaS is not perfect, tough [9]. In general, FaaS is stateless — as the execution environment may be removed at any time, data has to be persisted on external storage (like a network file system, object storage, or a database). Moreover, FaaS execution is not transactional – if a function is interrupted in the middle of the execution (e.g., due to reaching the resource limit), all changes done in remote object storage will remain.

Currently, FaaS implementations differ between cloud providers. They offer different predefined environments and impose different resource limits, e.g., maximum request processing times. Some implementations (like Microsoft Azure) introduce a concept of *application* – a set of functions deployed and running together. Nevertheless, all implementations share common principles: cloud users can define a function by selecting one of the supported languages; a function is executed in an environment isolated from other users' function environments; and the number of currently running environments is controlled by the cloud provider.

1.2 Related work

Serverless computing is a rapidly evolving cloud computing model with a wide range of possible applications. In this section, we present a general view of the current state of research on serverless computing, its applications and related problems. Additionally, in further chapters, we provide references to works directly related to analyzed topics.

An insightful analysis of benefits and current challenges in serverless computing is presented in [10]. Authors believe that serverless differs to a great extent from previous cloud computing models. They indicate that in serverless computing, cloud customers are not required to manage resource allocation on their own. Also, cloud customers can use serverless computing without expert knowledge of cloud architecture. As the cloud provider scales automatically, the end-user experience can improve, compared to a scenario when an application is running on a fixed number of instances or scaling takes significant time. Moreover, cloud users are billed for actually used resources. This approach can lead to potential cost savings, especially if a function is invoked only occasionally. Authors also note that serverless applications are usually written in a high-level language (like Python or JavaScript), which allows the cloud provider to use different CPU architectures interchangeably.

However, in some applications, serverless computing is still inefficient. Prominent examples include HPC workloads [11], like large-scale linear algebra computations. Also, serverless computing often suffers from latency problems — for example, in the FaaS model the data has to be retrieved from external storage (e.g., object storage). Further analysis of performance challenges connected with FaaS is presented in [12].

Another issue is connected with a potential vendor lock-in — while all major cloud providers have a serverless offering, the offered services differ significantly so the deployed applications have to be adapted to a particular cloud provider. A recent overview [8] predicts that a common standard will be established as the serverless model becomes more popular.

While serverless computing is a relatively new paradigm (e.g., AWS Lambda was initially released in 2014), this is an active research area — a recent survey [13] analyses 275 papers. In the remainder of the short survey here, we focus on papers that optimize resource management and address problems with the adaptation of serverless computing. To improve readability, we propose a classification based on the core topic they focus on.

Adopting existing systems to the FaaS model. While the FaaS model enables rapid prototyping and development of new applications, the existing cloud applications require adaptation before they can be executed in this model. There is ongoing work to ease the migration process: M2FaaS [14] demonstrates *FaaSification* — automated porting of existing Node.js monolith applications to the Function as a Service model. Therefore, some of the mature systems can instantly benefit from the scaling and pricing model provided by FaaS model.

Nevertheless, as the FaaS model significantly differs from the "classic" cloud computing models, not all cloud computing use cases can be easily formulated in the FaaS model. Migration process may require careful manual porting to ensure that all requirements are fullfilled. In [15] authors present FaaS implementation of coordination service providing the same consistency guarantees and interface as ZooKeeper [16]. ZooKeeper is used by distributed applications for synchronization and configuration management. Serverless adaptations of services like ZooKeeper enable complex systems to be ported to the serverless computation model.

Executed functions may interact with the environment by modifying the shared state. In current FaaS platforms, interruption of a function call may result in an inconsistent state. [17] presents a shim enabling fault-tolerance and benchmarks its impact on the performance. Despite being a primarily cloud computing model, FaaS can be also used on classic supercomputers. In [11] authors study the benefits of using serverless computing for MPI-based high-performance applications.

Assigning calls to nodes. In a typical FaaS architecture, incoming invocations are dispatched onto worker nodes – each function call is assigned to a single worker node. Then, this worker node is responsible for managing the execution of a call, including the creation of the required environment (when there is none available). The process of assigning a call to a worker node is one of the key points in the FaaS processing pipeline where optimizations are applied.

In [18], a central scheduler assigns requests to individual cores (rather than nodes, as in the standard OpenWhisk [19]). FaaSRank [20] optimizes the assignment of the invoker to a function call through a neural-network approach. Further optimizations are possible for complex FaaS invocations (in which a function calls another function forming a chain of invocations). Fifer [21] optimizes the execution of such chains while avoiding coldstarts by reactive scaling. Fifer implements its prediction model for incoming invocations using LSTM. [22] analyzes DAGs and shows a decentralized scheduler reducing DAG processing time. Faastlane [23] speeds up the processing of workflows by executing multiple steps within a single executor in parallel while providing isolation between steps through Intel Memory Protection Keys (MPK).

Reducing cold starts. Creating an environment can be a resource- and time-consuming process, especially when compared to execution of a single call. During environment initialization, system resources have to be acquired and, on the first run, required files and data might have to be downloaded. As systems resources are limited, unused environments often have to be removed. Therefore, the reduction of cold start duration is an important issue. [24] reduces cold start time by restoring a function instance from a predefined state (authors present an implementation based on Google gVisior [25] sandbox). Particle [26] identifies network provisioning as an important factor influencing startup time in platforms using containers. To address this issue, the proposed solution decouples the creation of a network from the container creation process and uses a pool of ephemeral IPs. SEUSS [27] uses unikernel snapshots to reduce the time of initialization by over an order of magnitude and additionally saves memory by sharing pages between instances. [28] decouples libraries and other dependencies from function packages; libraries are cached on worker nodes, and the scheduler is aware of the packages available on a node. Such an approach leads to the reduction of startup time as large libraries do not have to be downloaded multiple times. [29] analyzes functions with similar dependencies — this allows a function call to use an environment dedicated for another, related function. [30] uses reinforcement learning to identify function call patterns and start containers in advance. Another approach [31] uses WebAssembly as a function runtime engine reducing cold start latency by up to 99.5%. All these approaches require provider-side modifications. In contrast, [32] proposes customer level middleware to be deployed with function code that can trigger container creation in advance.

Reducing resource requirements. As node resources are limited, a limited number of environments can co-exist on a single node. Thus, reducing the resource requirements of each environment can lead to overall performance improvements. Photons [33] share context (runtime, libraries, etc.) between multiple concurrent calls, thus reducing the memory footprint. OFC [34] uses ML to predict the true memory usage of an invocation; and then overcommits the rest to act as a cache for a remote data store. SAND [35] distinguishes between applications and functions; multiple functions of the same application share a common container. [36] extends FaaS to handle stateful functions natively, rather than through a remote data store.

Node level scheduling. While the FaaS cluster is built of dozens to thousands of nodes, some optimizations can be implemented on the level of a single node. As these methods do not involve inter-node communication, they do not impact nor depend on the data center network. ETAS [37] proposes a queuing policy similar to EECT proposed in Chapter 5. Although this policy is also implemented in OpenWhisk, the impact of OS-level preemptions is diminished, and thus the potential of reducing their number is untapped. In contrast, we propose container-management methods that address this issue. In [37], the policy is tested in an underloaded system against a set of 4 custom functions (and no I/O-bounded functions), rather than a standard benchmark we used. Also, no fairness of function executions is considered (in contrast to our FC policy presented in Chapter 6).

SFS [38] addresses the impact of OS preemption on function execution. The proposed solution changes Linux scheduling class of function processes from standard CFS scheduler to FIFO (preventing preemption) for limited time. This way, short function calls may finish without interruption, while long calls will not block a CPU core for too long.

Models of resource management In the search for the FaaS resource management model, we adopt classic, ubiquitous models used in resource allocation-related problems, such as bin packing or scheduling.

In the bin packing problem [39], a set of items of given sizes has to be placed into a finite number of fixed-size bins. The objective is to minimize the number of used bins. There are numerous variants of this problem. For instance, in vector bin packing [40] the bins and objects can have multiple dimensions. In the online bin packing [41] items are not known in advance and have to be placed as they arrive. In a stochastic variant [42] of bin packing, sizes of items are described by a distribution and the probability of overflow bin's capacity is considered. In FaaS model, placement of the function environments can be represented as the multi-dimensional, online bin-packing – as the FaaS system process incoming load, we become aware of required environments that have to be created on available machines. In Chapter 3 we further explore the problem of packing function environments.

The scheduling problem [43] considers a set of jobs and a list of machines. The goal is to produce a *schedule* – an assignment of jobs to machines, which satisfies given restrictions. Fundamentally, a machine cannot process more than one job at a time. Additionally, we may require that a job has to be processed on a particular subset of machines or cannot be started before its *release time* [44]. The scheduling theory considers a number of scheduling problem types. The variants applicable to FaaS include scheduling with setup times [45] (preparation time required for machine setup before a new type of task can be processed), parallel machines [46] (every machine can process every job), and scheduling with precedence constraints [47]. Moreover, in FaaS we can also consider both preemptive (when job execution may be paused and resumed later) and non-preemptive cases. A book [48] presents a comprehensive comparison and classification of scheduling problems. A survey [49] analyses multiple variants of scheduling problems with setup times or costs. Furthermore, in Chapter 3 we consider scheduling with setup times and precedence constraints and present a comprehensive analysis of papers related to this model.

In the divisible load [50] scheduling model the workload can be divided into parts that can be processed in parallel. Thus, our goal is to minimize processing time, by splitting load onto multiple parallel processors and optimizing the usage of available computing power. There are multiple variants of this model, e.g. loads may be *arbitriaily divisible* or *modularly divisible* if there are constraints on resulting partial loads. This model is straightforwardly applicable to FaaS computations: in the FaaS model, the invocations ("load") are triggered in response to incoming events (e.g. HTTP requests). Then, the invocations are scheduled onto function execution environments running on worker machines. Moreover, this pattern is similar to other scenarios in cloud computing, in which we distinguish two components: (1) a (horizontal) autoscaler (e.g. [51]) that adds or removes application instances in response to long-term changes; and (2) a load balancer (e.g. [52]) that dynamically assigns incoming requests to running instances.

1.3 Dissertation structure and results

As Function as a Service is a relatively new paradigm, there are many aspects of possible optimizations. In this work, we focus on improvements in scheduling and resource management. Our goal is not to lead to (yet another) FaaS interface, but to indicate improvements that can be applied to existing systems. We model existing systems — both from the cloud provider and cloud user perspective – and we aim to ensure that proposed changes can be applied without the need to re-design the cloud infrastructure or the deployed applications. While computing models change over time, the current data center architecture is the result of years of development [53]. Thus, we assume that limitations resulting from the data center architecture are well-justified.

In the following chapters, we analyze possible optimizations. Internally, chapters keep a consistent structure – we start with the definition of the scheduling model and conclude by discussing related work and summarizing the main results.

In **Chapter 3** we analyze a recent addition to FaaS – the ability to compose functions: a function may call other functions, which, in turn, may call yet another function – forming a directed acyclic graph (DAG) of calls. 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 the necessary preparation of environments, thus reducing the impact of cold starts. We model resource management in composite 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 derived from a trace from Microsoft Azure. Our results show that if the setup times are long compared to invocation times, algorithms that use information about the composition of functions and their setup times consistently outperform greedy, myopic algorithms, leading to a decrease in the average response latency by at least a factor of two.

In **Chapter 4** we analyze a model that captures (semi)-flexibility of cloud resource management. Cloud resource management is often modeled by two-dimensional bin packing with a set of items that correspond to tasks having fixed CPU and memory requirements. However, applications running in clouds are much more flexible: modern frameworks allow to (horizontally) scale a single application to dozens, even hundreds of instances; and then the load balancer can precisely divide the workload between them. This is particularly convenient for the FaaS model, as the cloud customers specify only the code and environments and the cloud provider is entirely responsible for scaling environments and load balancing incoming calls. Each cloud application (function environment) is characterized by its memory footprint and its momentary CPU load. Combining the scheduler and the load balancer, the resource manager decides how many instances of each application will be created and how the CPU load will be balanced between them. In contrast to the divisible load model, each instance of the application requires a certain amount of memory, independent of the number of instances. Thus, the resource manager effectively trades additional memory for a more evenly balanced load. We study two objectives: the bin-packing-like minimization of the number of machines used; and the makespan-like minimization of the maximum load among all the machines. We prove NP-hardness of the general problems, but also propose polynomial-time exact algorithms for boundary special cases. Notably, we show that (semi)-flexibility may result in reducing the required number of machines by a tight factor of $2 - \varepsilon$. For the general case, we propose heuristics that we validate by simulation on instances derived from the Azure trace.

In **Chapter 5** we take the perspective of a single node in a FaaS cluster. We assume that all the execution environments for a set of functions assigned to this node have been already installed. Our goal is to schedule individual invocations of functions, passed by a load balancer, to minimize response time and related metrics. Deployed functions are usually executed repeatedly in response to multiple invocations made by end-users. Thus, our scheduling decisions are based on the information gathered locally: the recorded call frequencies and execution times. We propose a number of heuristics, and we also adapt some theoretically-grounded ones like SEPT or SERPT. By simulations, we show that, compared to the baseline FIFO or round-robin, our data-driven scheduling decisions significantly improve the performance.

In **Chapter 6** we take a system perspective on scheduling on a single worker node. Inspired by results obtained from simulations in Chapter 5, we extend OpenWhisk, an open-source FaaS system, and implement new scheduling algorithms. We measure the efficiency of the proposed solutions by experiments using workload from SeBS [54] benchmark. In a loaded system, our method decreases the average response time by a factor of 4. The improvement is even higher for shorter requests, as the average stretch is decreased by a factor of 18. This leads us to show that we can provide better response-time statistics with 3 machines compared to a 4-machine baseline.

1.4 Papers covered in the Thesis

The preliminary results were published in the following papers:

- P. Żuk and K. Rzadca, "Scheduling Methods to Reduce Response Latency of Function as a Service", in *32nd IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2020, Porto, Portugal*, IEEE, 2020, pp. 132–140 (results in Chapter 3)
- P. Żuk and K. Rzadca, "Reducing response latency of composite functions-as-a-service through scheduling", *Journal of Parallel and Distributed Computing*, vol. 167, pp. 18–30, 2022 (results in Chapter 3)
- B. Przybylski, P. Żuk, and K. Rzadca, "Data-driven scheduling in serverless computing to reduce response time", in *IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 206–216 (results in Chapter 5)
- B. Przybylski, P. Żuk, and K. Rzadca, "Divide (CPU Load) and Conquer: Semi-Flexible Cloud Resource Allocation", in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, IEEE, 2022, pp. 129–139 (results in Chapter 4)
- P. Żuk, B. Przybylski, and K. Rzadca, "Call Scheduling to Reduce Response Time of a FaaS System", in *IEEE Cluster*, IEEE, 2022 (results in Chapter 6)

Chapter 2

Preliminaries

In this chapter, we introduce common concepts of the Function as a Service model. As in this work we analyze multiple aspects of resource allocation and possible optimizations, in Section 2.1, we introduce commonly used symbols and notation. In Section 2.2, we describe Apache OpenWhisk — an open-source serverless platform that we use as a inspiration for our theoretical models and a core build block in our experiments.

2.1 Common definitions and notation

A *function* is a code snippet along with a specification of a required execution environment (required compiler or interpreter, libraries, etc.). By a function *call* we understand a single execution of the function code. Functions do not have state – while it is technically possible to store data in the environment (e.g., by creating a file), developers must not rely on this possibility as environments may be created and destroyed between the calls. Therefore FaaS in the cloud is often used with other services providing a persistence layer, such as an object storage service or a database.

A typical FaaS system is composed of many *worker nodes* and a *controller* (which may be replicated to ensure high availability). The controller receives incoming events (e.g., HTTP requests), maps them to the appropriate function and schedules their execution.

A worker node is responsible for spawning and maintaining execution *environments*. If a function call triggers the creation of a new environment, we call such event a *cold start*. Creating a new environment usually takes a non-negligible amount of time ([60] reports at least 500 ms) and should be avoided. However, unused environments still consume resources (e.g., RAM, disk storage), thus creating a new environment may require an *eviction* of the existing one.

This work draws from a scheduling theory. A single call of function maps to a *job*, and a single CPU core is represented as a *processor*. In this work, we consider both the *clairvoyant* model (when the scheduler has complete information about the arrival time of executed jobs, their processing times, and resource requirements) and models with limited clairvoyance. Similarly, the scheduling algorithms can be split into *offline* and *online* – the former has access to the whole input at the beginning (all jobs to be scheduled are known), while the latter group has a limited view (e.g., the new jobs arrive over time). Our notation follows the standard of Brucker [48]. We consider uniform worker nodes with m number of parallel processors/cores: $P_1, P_2, \ldots, P_m$. We denote a set of n stateless functions as $f_1, f_2, \ldots, f_n$. Each function f_j can be executed multiple times. For i-th call, we denote the moment of the call by $r(i)$, completion time by $c(i)$, processing time by $p(i)$, and setup time by $s(i)$ [61], [62].

2.2 An architectural overview of Apache OpenWhisk, an open source FaaS system

While popular cloud providers have serverless platforms in their offer, there are also notable examples of open-source platforms. These platforms are particularly interesting from a cloud researcher's perspective, as their internal components can be modified, which is essential in experimental evaluation. In this section, we describe from the resource management perspective a representative implementation of a serverless cloud platform, the open-source Apache OpenWhisk [63]. We focus on Apache OpenWhisk as it is mature, actively-developed software also offered commercially (IBM Cloud Functions, Adobe I/O Runtime). OpenWhisk features include support for chaining functions and multiple parameters that can be tuned (like per-function time limits). OpenWhisk alternatives include OpenLambda [64] and Fission [65]. OpenLambda uses containers to provide a runtime environment for functions. Fission is designed for Kubernetes [5]; it can be deployed on existing cluster among other applications, which ease its introduction.

OpenWhisk allows a cloud *customer* to upload *functions'* code and specify Docker image to be used to create environments. OpenWhisk executes stateless functions (also called *actions*) in response to events triggered by HTTP requests, fixed alarms or other functions. OpenWhisk isolates function by invoking a call in a container initialized with a runtime environment specific to a programming language in which the function was defined (a function can also request a customized Docker image). One of the key roles of OpenWhisk is to manage the calls and the containers as the load of the deployed functions may dynamically change.

Before the first execution of a function, the environment must be initialized (e.g., setting up the container, compiling function code, or installing dependencies). This initialization can take a considerable amount of time and we call it later the *setup time.* An environment is specific to a function

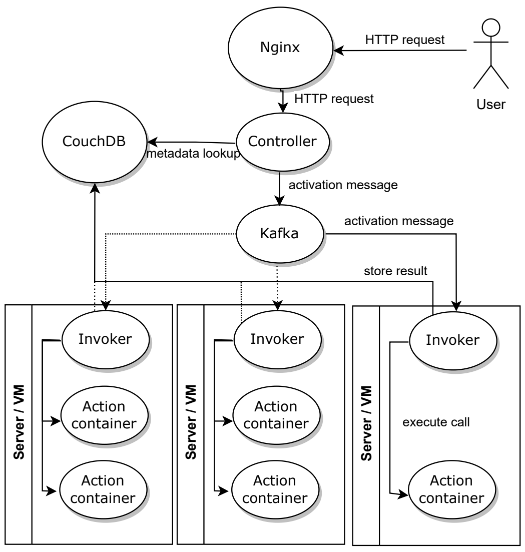


Figure 2.1: Core architecture of OpenWhisk

— after setup, a particular environment is not reused by different functions. However, subsequent invocations of the same function 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 composing of 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. 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.

Figure 2.1 presents a high-level overview of OpenWhisk internal modules. From our perspective, the key components are the *controller*, the *invokers*, and the *action containers*. Internally, OpenWhisk is written in Scala [66] with the usage of Akka [67] toolkit. The system uses actor-based concurrency, and most of the processing is event-driven. Separation of the components enables scalability of the system – in particular, invokers may be added to increase cluster computing capacity. The *controller* and *invoker* components use a publish-subscribe system (Apache Kafka) to discovery and communicate. Communication between all remaining components uses HTTP, following the standard client-server pattern.

The invoker is an agent running on a single *worker node* and acting as its node-level resource manager. The invoker is responsible for executing actions scheduled on the managed node. Each invoker has a unique identifier and it announces itself to the controller while starting. A node hosts multiple action containers, which in turn execute function calls and return the results.

The invoker is also responsible for assigning a function call to a specific action container. In the standard scenario, when a new request arrives to the invoker and there are any pending requests, the request is added to the queue. Otherwise, the invoker tries to arrange a container on which the request can be executed immediately. First, the invoker tries to locate a *free pool container* matching the request. A free pool container is a container that is initialized with the execution environment and the requested function. Such containers can usually be found if a particular function is executed repeatedly. If a free pool container is not available, the invoker tries to find a *prewarm pool container.* A prewarm pool container is initialized with the execution environment suitable for a group of functions (e.g., all Python functions), but the specific function has not yet been initialized. If no free or prewarm containers match the request, the invoker tries to create a new container. This may be impossible if there is not enough free memory: in such a case, some non-matching free pool containers may be removed (evicted). Finally, if there are not enough free pool containers to be evicted, the action call is queued in a FIFO (*First-In, First-Out*) queue.

The controller acts as a scheduler handling incoming events and attempts to balance load across nodes by routing each function invocation to a concrete invoker. The controller also monitors the status of workers and the currently executing invocations. In the standard OpenWhisk, 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.

Although simple to implement, balancing the load in this method is more static, as the decision to execute a request at a certain invoker cannot be easily reversed (and, if the invoker fails, the assigned requests are lost). To counteract that, OpenWhisk is currently gradually switching to a new action assignment model [68] based on global queues. In this new model, the controller does not decide which invokers will be responsible for executing actions. Instead, each invoker pulls requests from common Kafka queues: each function has its own global queue, and the invoker pulls requests from queues matching its free pool containers. However, the controller still decides what kinds of containers will be created by specific invokers.

Chapter 3

Scheduling composite functions

While FaaS is gaining wide adoption, a recent new element is still relatively unexplored — the *composition* of functions [69]. 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.

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

This chapter is structured as follows:

- We describe and analyze the model of scheduling in FaaS as a combination of the multiple knapsack problem, scheduling with dependencies and with setup times (Section 3.1).
- We propose a number of heuristics for each aspect (Section 3.2). These heuristics derive from classic approaches, but we adjust them to the FaaS model.
- By simulations, we show that heuristics examining the composition structure lead to lower response latencies (Section 3.3).
- We analyze related work with focus on function composition (Section 3.4).
- We summarize our work and discuss results (Section 3.5).

3.1 A scheduling model of FaaS composition

In this section we define the optimization model for the composite 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 [48]. In this chapter extend the notation described in Section 2.1 as follows. A single end-user request corresponds to a *job* J_i. A job is composed of one or more *tasks* (calls) $O_{i,k}$, 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's $O_{i,k}$ 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 call $O_{i,k}$ to exactly one *function* $f(O_{i,k})$ (obviously, two tasks $O_{i,k}$ and $O_{i,l}$ from a job J_i might belong to different functions). All tasks of a function 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 call $O_{i,k}$ of a function $f(O_{i,k})$ is executed on exactly one machine in an *environment* (OS container) $E(f(i))$. $E(f(i))$ requires set-up time $s(f(i))$ (initialization of the environment) before executing the first task. Subsequent calls 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 an 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 [70].

In this model 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 also 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 [71]).

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 that follow the first task of a job cannot be scheduled at time 0, 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). [51] 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 and this environment 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)
                    ▷ $policy \in \{default, start\}$, $wait \in \{true, false\}$
    for $task \in$ FINISHEDTASKS($t$) do
        if $policy == default$ then
            QUEUEDEPENDENTTASKS($task, t$)
    for $task \in$ ORDER($queue$) do
        $e \leftarrow$ FINDUNUSEDENVIRONMENT($task$)
        if $e$ is nil and $wait$ then
            $e \leftarrow$ FINDENVIRONMENTTOWAIT($task$)
        if $e$ is nil then
            $e \leftarrow$ PLACENEWENVIRONMENT($task$)
        if $e$ is nil then
            $e \leftarrow$ REMOVEANDPLACEENVIRONMENT($task$)
        if $e$ is not nil then
            ASSIGNTASK($c, task$, RELEASETIME($task$))
            REMOVEFROMQUEUE($task$)
            if $policy == start$ then
                $p \leftarrow$ DURATION($task$)
                QUEUEDEPENDENTTASKS($task, t + p$)
```

This problem is NP-hard, as generalizing several NP-hard problems (bin-packing [72], $P2|DAG|\sum C(i)$ [48]). 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 has at most one environment); and having each tasks in a separate family, all setup times eual to 0, and processing times $p(f)$ equal to processing times of tasks in the $P2|DAG|\sum C(i)$ instance.

3.2 Heuristics for scheduling invocations

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 3.2.1) and the awareness of task dependencies (Section 3.2.4). *Deployment* corresponds to the removal policy (Section 3.2.2). *Allocation* corresponds to the waiting/non-waiting variants (Section 3.2.3).

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 (thus completion times of all dependencies are known), along with their *release time* (*queueDependentTasks*). We maintain information about a 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 in Section 3.2.4).

2. Apply a scheduling policy to the queued tasks (*Order*). We describe policies for this step in Section 3.2.1.

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 3.2.3 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 3.2.4).

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., $(FIFO, LRU, wait, start)$.

3.2.1 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.

3.2.2 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 the environment popularity:

- LRU: remove the least recently used (LRU) environment(s) from the first machine 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.

3.2.3 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 [73], [74]) that places a task on a processor that will finish the task as the earliest.

3.2.4 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).

3.3 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 different machine environments; Section 3.3.1 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 3.3.2 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.

3.3.1 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 3.3.9, we adopt a recently-published Microsoft Azure Trace [75] 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 a 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).

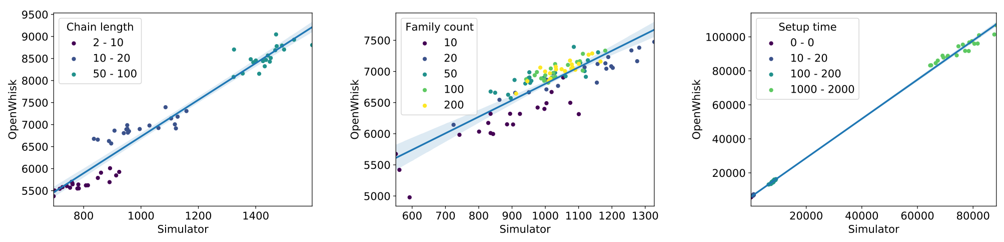
We consider two sets of test instances: DAGs and chains. While DAGs fully express function compositions, we consider chains as an important case as they are directly supported by OpenWhisk platform — therefore results of our research can be applied to a real system. Moreover, chains enable us to validate our simulator against OpenWhisk (Section 3.3.2).

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 a 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, $(1/n) \sum C(i)$. We omit results on tail, 95%-ile latency – the 95%-ile



(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 ($n_f = 50$, $l \in [10, 20]$)

Figure 3.1: Average latency on OpenWhisk system (Y axis) and simulated OW policy (X axis) with linear regression model fit. 1 unit is 10ms. Each point corresponds to a single test instance executed on both OpenWhisk and simulator. Translucent bands indicate the 95% confidence interval.

results are analogous to the averages(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 2.2) with an algorithm *OW*. *OW* randomly selects for each family f the initial machine m_f and the *step size* k_f, an integer co-prime with the number of machines m. When scheduling a task $O_{i,k}$ in family f, *OW* checks machines m_f, $m_f + k_f$, $m_f + 2k_f$, $\ldots$ (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.

3.3.2 Validation of the simulator against OpenWhisk

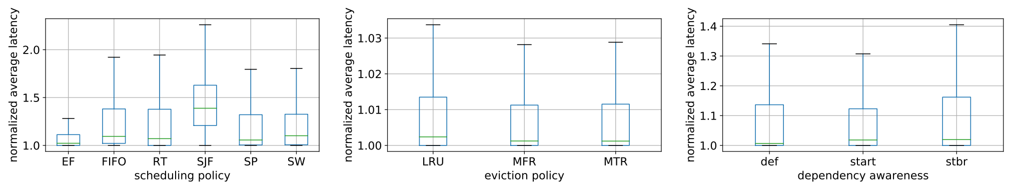
To compare the results of our simulator with OpenWhisk, we developed a customized OpenWhisk execution environment that emulates a function with a certain setup time $s(f)$, execution time $p(f)$ and resource requirement $q(f)$. We chose 10ms as the time unit to reduce impact of possible fluctuations of VM or network parameters in the datacenter (we performed some early experiments with 1ms and this noise was significant; and with a longer time unit tests take unreasonable time). This environment emulates initialization by sleeping for $s(f) * 10ms$; and it emulates execution by sleeping for $p(f) * 10ms$. While sleeping does not use the requested memory ($q(f) * 128MB$), the memory is blocked and therefore cannot be simultaneously used by other environ-

Table 3.1: 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. 3.1

Group	5th percentile	median	95th percentile
Fig 3.1(a)	0.86	0.93	0.97
Fig 3.1(b)	0.29	0.67	0.84
Fig 3.1(c)	0.997	0.998	0.9995

ments. 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 16GB RAM, and were running Ubuntu 18.04 LTS. We further restrict the memory OpenWhisk can use on machines to 1280MB (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 Figure 3.1 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* [76] 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. 3.1.b, and 0.999 when varying the setup time, Fig. 3.1.c). To further test our claim, we compute the coefficients of determination (R^2) scores [77] 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 3.1. Thus, the R^2 scores are approximately equal to the squares of the Pearson correlation coefficients.



(a) (b) (c)

Figure 3.2: 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 times Inter-Quantile Range (IQR).

There is, however, an additive factor in OpenWhisk noticeable especially in smaller test instances in Fig. 3.1.(a) and Fig. 3.1.(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. 3.1.(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.

3.3.3 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 a 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. Figure 3.2 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).

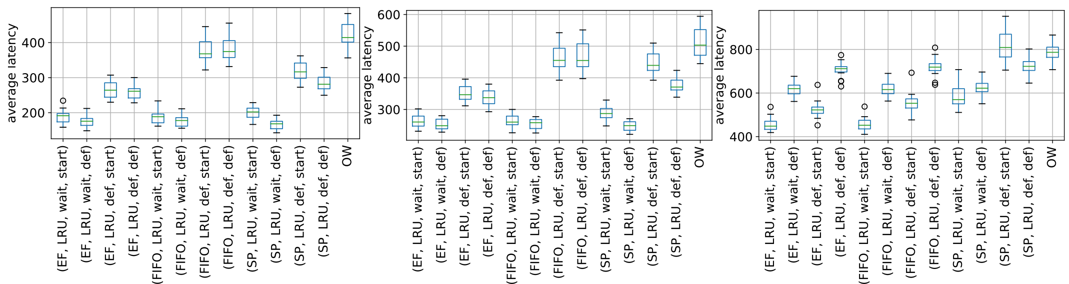
Ordering: *EF* policy dominates other ordering policies, confirming that it is better to avoid environment setup by reusing existing environments. Its median is similar to RT (and lower than other algorithms), and the range of values (including the third quartile) is the lowest.

Eviction: 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 (Figure 3.2), 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 only SP. Finally, as the difference between *start* and *stbr* variants is small, we show results only for *start*.

3.3.4 Impact of the number of tasks in a job



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

Figure 3.3: 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.

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 a 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 Figure 3.3, 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.

3.3.5 Impact of the number of families

Figure 3.4 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. 3.4.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.

3.3.6 Impact of the setup time

Figure 3.5 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 re-creation. 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

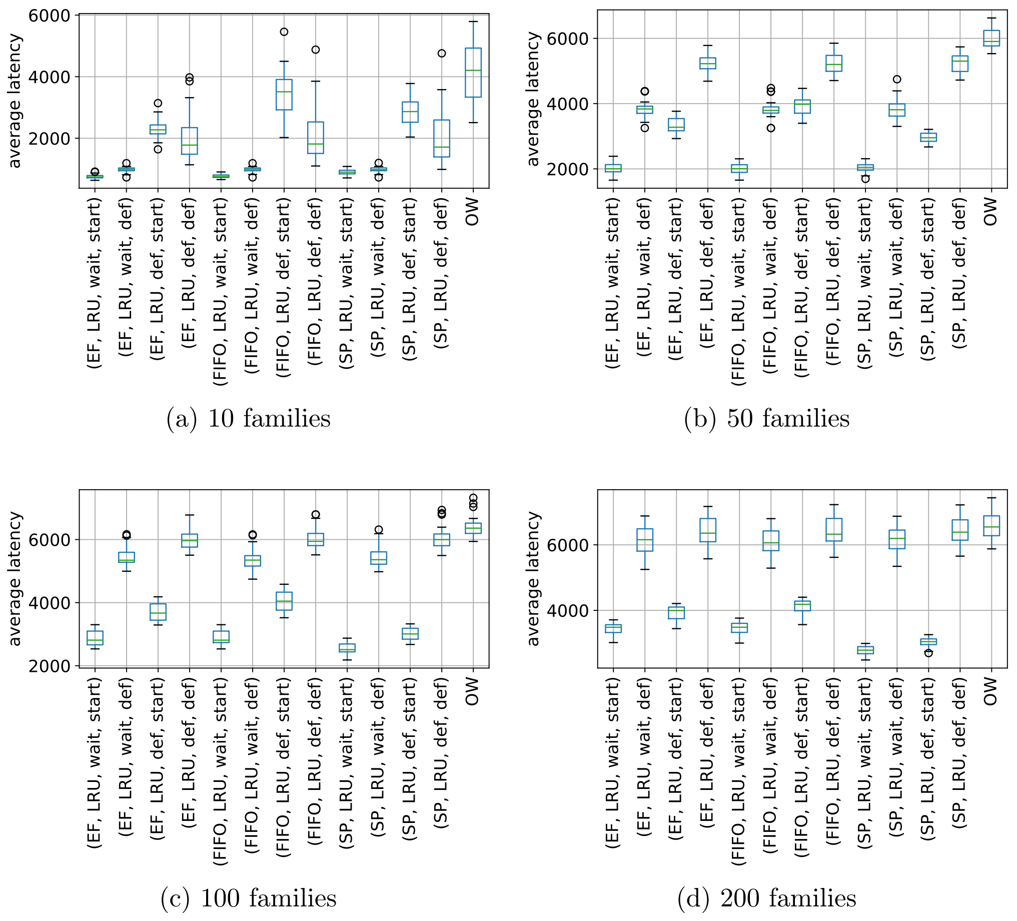
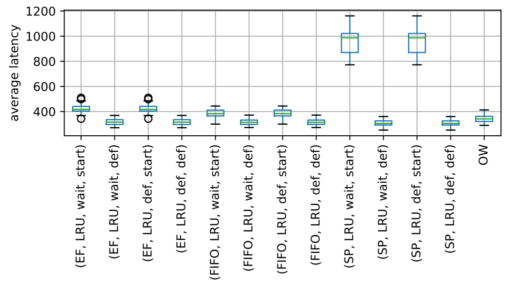
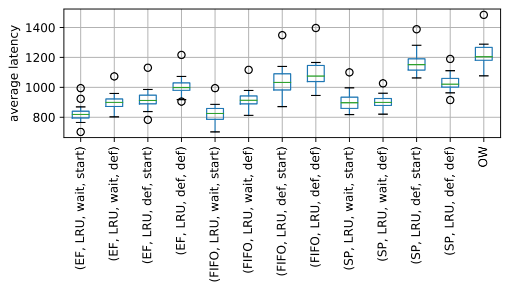


Figure 3.4: Influence of the different number of families. To show general trend, we present results for 10, 50, 100 and 200 families. For all test instances $m = 20$, $Q = 10$, $s(f) \in [100, 200]$, $l \in [50, 100]$

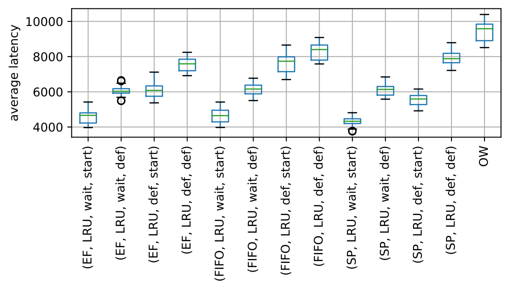
longer. This behavior is caused by adding tasks with future release time to the queue (see Section 3.2.4). Consider two jobs each of two tasks: 1. a *long* job with task A (duration 10) followed by task B (duration 1); 2. a *short* job with task C (duration 1), followed by task B (duration 1). *EF* and *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).



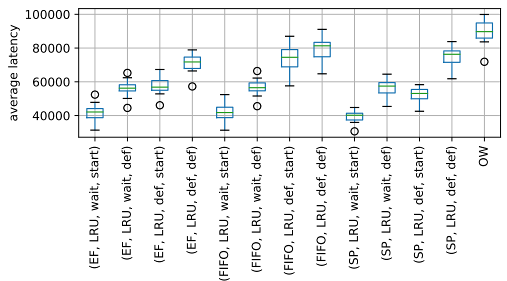
(a) setup time 0



(b) setup time 10-20



(c) setup time 100-200



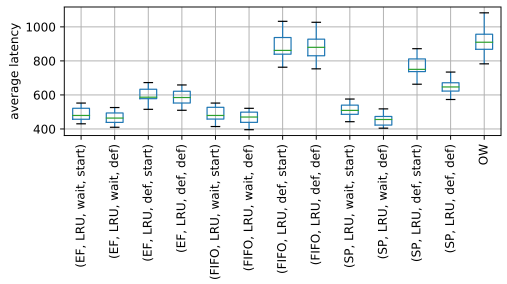
(d) setup time 1000-2000

Figure 3.5: Influence of the setup time. For all test instances $n_f = 50$, $m = 10$, $Q = 10$, $l \in [50, 100]$.

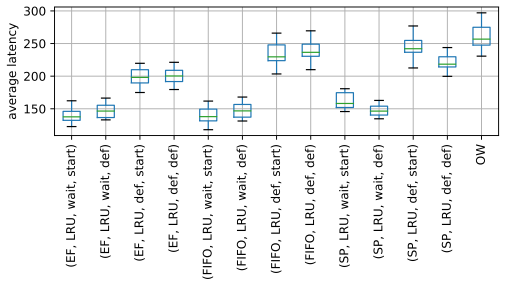
We further investigate for which test instance parameters the dependency-aware *start* dominates the myopic *def*, assuming non-negligible setup times $s(f) \geq 100$. We aggregate results by all simulation parameters (count of families n_f, machines m, machine capacities Q, range of job sizes l, range 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 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.

3.3.7 Impact of machine capacity

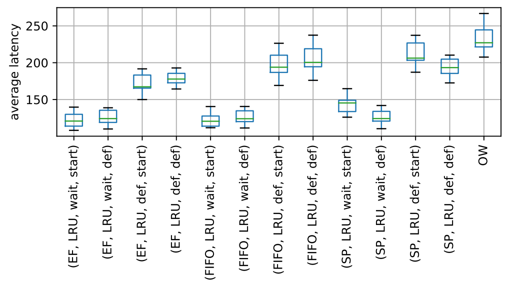
Figure 3.6 compares results as a function of the number of machines and their capacity. For all test instances $n_f = 50$, $l \in [10, 20]$, $s(f) \in [10, 20]$. To show a general trend and ensure clarity, out of 15 considered machine configurations, we present results only for test instances with $(m, Q) \in \{(5, 20), (20, 20), (50, 10), (50, 50)\}$. For cases up to $(m, Q) = (5, 20)$, the only



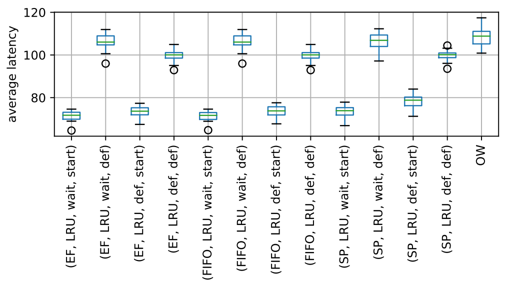
(a) 5 machines, capacity 20



(b) 20 machines, capacity 20



(c) 50 machines, capacity 10



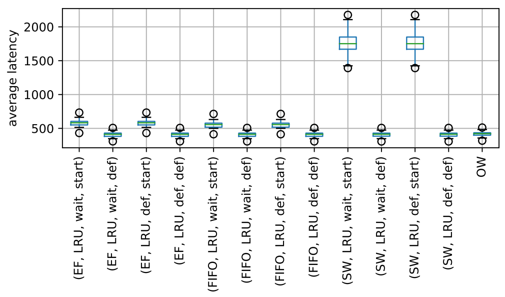
(d) 50 machines, capacity 50

Figure 3.6: Influence of the machine environment. For all test instances $f_n = 50$, $l \in [10, 20]$, $s(f) \in [10, 20]$

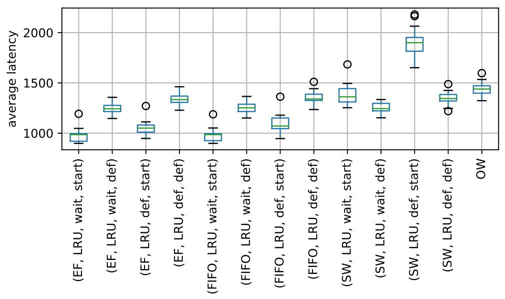
non-negligible differences between the plain and dependency-aware variants are for SW and SP scheduling policies. Due to large number of jobs (job sizes are in range 10-20), when dependent tasks are added to the queue earlier, environments may get blocked as described in Section 3.3.6, therefore there 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.

3.3.8 Differences between DAGs and chains

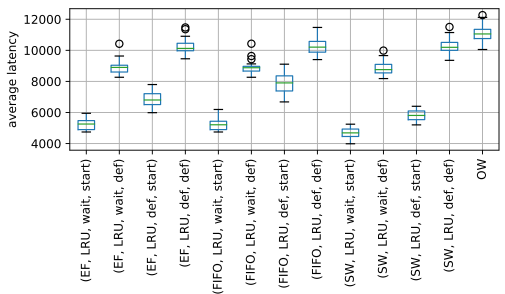
We also performed analysis analogous to Section 3.3.6, Section 3.3.4 and Section 3.3.7, but for chains, instead of DAGs. Chains are particularly inter-



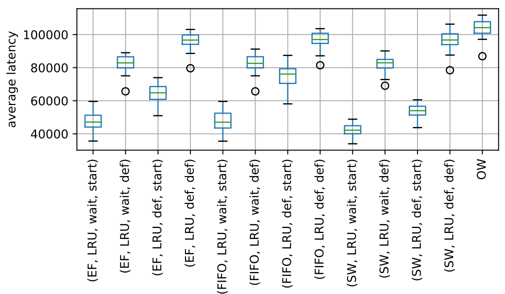
(a) setup time 0



(b) setup time 10-20



(c) setup time 100-200



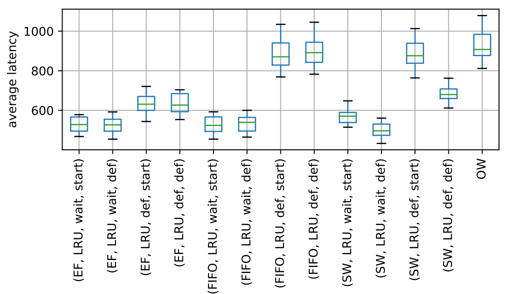
(d) setup time 1000-2000

Figure 3.7: Influence of the setup time for instances with chains. For all instances $n_f = 50$, $m = 10$, $Q = 10$, $l \in [50, 100]$.

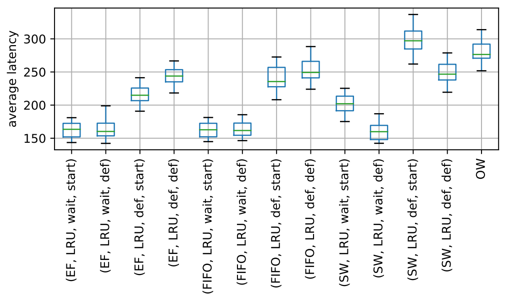
esting case as they are the simplest form of function compositions and they are sufficient to show the benefits resulting from better scheduling.

In general, obtained results are similar: if we compare behavior for different setup times (Figure 3.7 and Figure 3.5), we can observe that for non-zero setup times our algorithms perform better than baseline (OW). In both DAGs (Figure 3.6) and chains (Figure 3.8) 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.

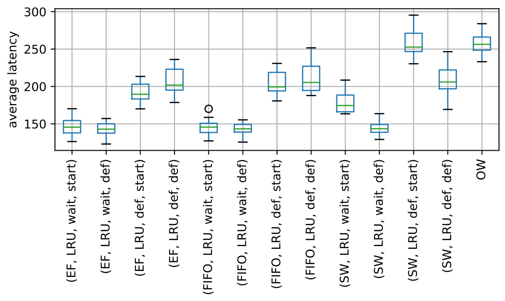
Nevertheless, there are observable differences. Increasing l, the number of tasks in a job, increases average latency more significantly for chains (Figure 3.9) than for DAGs (Figure 3.3). For the same number of tasks in a job, a chain has less available concurrency than a DAG (i.e., longer critical path). This also impacts differences observed in a comparison of dataset with different setup times (and constant job size) – for chains (Figure 3.7) we observe higher difference between variants without (*def*) and with depen-



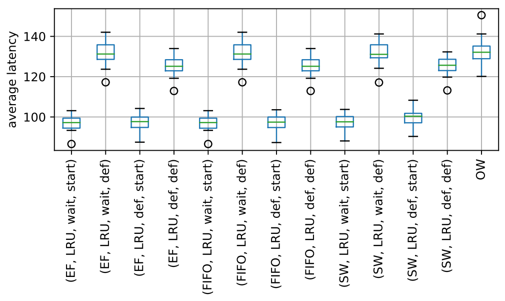
(a) 5 machines, size 20



(b) 20 machines, size 20

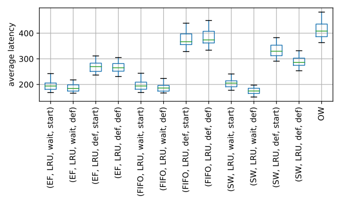


(c) 50 machines, size 10

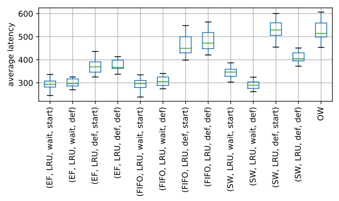


(d) 50 machines, size 50

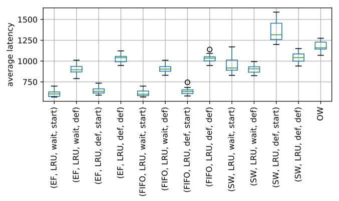
Figure 3.8: Influence of the machine environment for instances with chains. For all instances $f_n = 50$, $l \in [10, 20]$, $s_f \in [10, 20]$



(a) length 2-10



(b) length 10-20

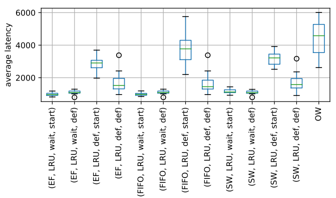


(c) length 50-100

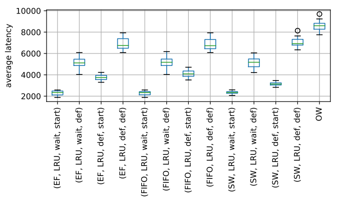
Figure 3.9: Influence of the length of the chain. For all instances $n_f = 50$, $m = 20$, $Q = 10$ with setup times 10-20.

dency awareness (*start*) than for DAGs (Figure 3.5). For similar reasons, for datasets with more families, our algorithms for DAGs (Figure 3.4) – as well as for chains (Figure 3.10)– provide lower average latencies than OW, but difference between the best schedule and the baseline for DAGs is noticeably smaller.

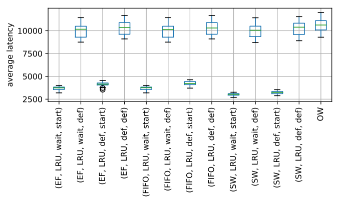
Overall, the results are similar, proving the robustness of our proposed algorithms. In all analyzed cases with non-zero setup times, our proposed



(a) 10 families



(b) 50 families



(c) 200 families

Figure 3.10: Influence of the different number of families for instances with chains. To show general trend, we present results for 10, 50 and 200 families. For all instances $m = 20$, $Q = 10$, $s_f \in [100, 200]$, $l \in [50, 100]$

algorithms – the *EF* scheduling policy with *LRU* replacement and *waiting* environment creation – outperform the OW baseline.

3.3.9 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 Microsoft Azure Trace [75] 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 handle 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 3.3.1, 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 3.3.1.

Figure 3.11 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.

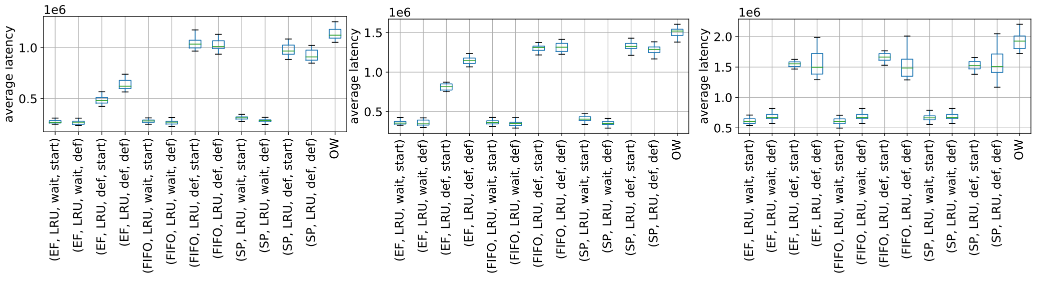
Figure 3.12 shows the impact of different setup times. Similarly to synthetic test instances, for configurations with negligible setup time, enabling dependency-awareness decreases performance. With considerable setup times (at least 100 times larger than the average duration), EF and FIFO with *waiting* and dependency-awareness (*start*) have one of the lowest average latencies.

In Figure 3.13 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 enable dependency-awareness (*start*) give significant improvement over baseline (OW).

3.4 Related work

Our model of FaaS resource management combines scheduling (with setup times and dependencies) [78] 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



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

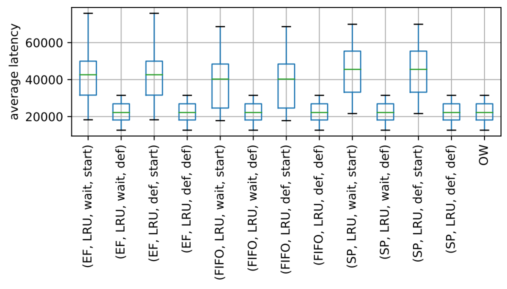
Figure 3.11: Comparison of different job sizes. In each case dataset contains 50 families generated from Azure trace. Setup times 100-200 times larger than average duration observed over all families, 10 machines of capacity 20.

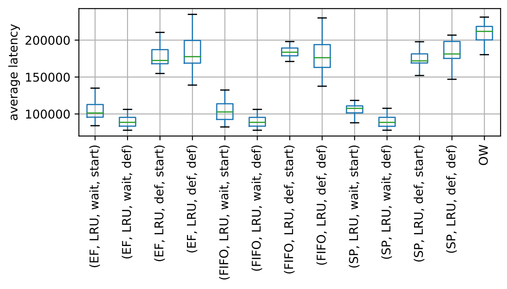
consistently dominated by our policies). Individually, these are classic problems in combinatorial optimization. Allahverdi [78] performs a comprehensive review of about 500 papers on scheduling with setup times. Brucker [48] reviews scheduling results. We start by describing the closest related combinatorial 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.

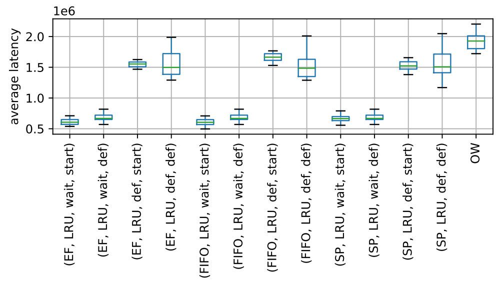
Quadratic programming: We proposed heuristics, rather than generic solvers or metaheuristics. Initially, we considered encoding our problem as an (integer) quadratic programming. Nevertheless, Gurobi [79] was unable to find an optimal schedule in 15 minutes (on a reasonable desktop machine) even for 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. [80] study similar problem of minimizing mean weighted completion time for tasks with sequence dependent setup times. [81] 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. [82] surveys workflow scheduling in the cloud. [83] measures how inaccurate runtime estimates influence the schedules which complements our study, as we assumed that estimates are known. [84] analyzes possible performance benefits of resource interleaving across the parallel stages. [85] proposes Balanced Minimum Completion Time, an algorithm for scheduling


(a) setup time 0


(b) setup time 10-20x avg. family duration


(c) setup time 100-200x avg. family duration

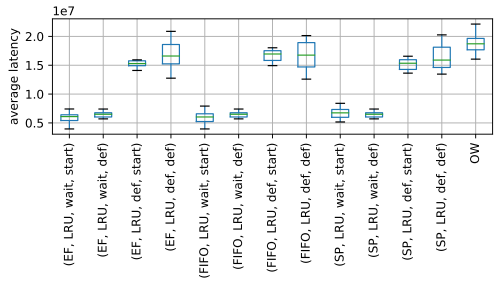
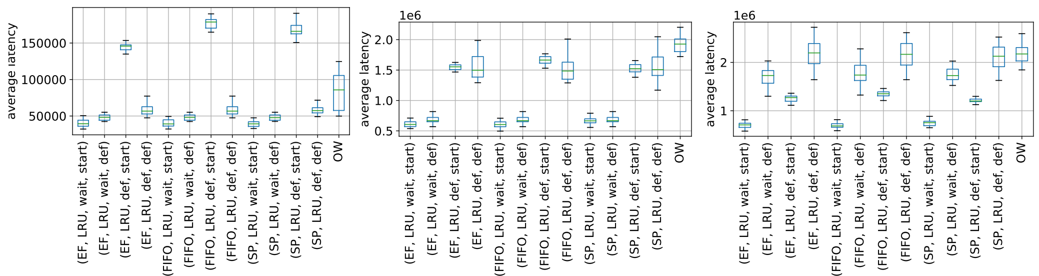

(d) setup time 1000-2000x avg. family duration

Figure 3.12: 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

tasks with dependencies (and without setup times) on heterogeneous systems. [61] 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 100s time limit for execution. Such long running times make this method unusable in data-center schedulers. [62] 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



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

Figure 3.13: 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)

we explore possible performance improvements by considering the composition of functions. [21], [22] also consider composed functions. [21] 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. [22] 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. [86] proposes checkpointing and then restoring environments. Catalyzer [24] reuses the environment state. Particle [26] 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. [87] proposes new isolation abstraction reducing memory requirements. Photons [33] reduces overall memory consumption by running multiple concurrent invocations within a single environment (without impacting reliability). ENSURE [88] 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,

	Q		
m	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) $s(f) \geq 100$

	Q		
m	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) $s(f) < 100$

Table 3.2: Average relative improvement in the average response latency of $(\cdot, \cdot, wait, start)$ over the OW baseline

the functions are stateless, getting the state from external storage takes time. Cloudburst [36] demonstrates and analyzes usability of stateful serverless computing. [89] presents framework for building stateful and fault-tolerant serverless-based applications, running on top of existing platforms.

3.5 Summary and discussion

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 chapter, 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.

We conclude the experiments over DAGs by presenting averages improvements of $(\cdot, \cdot, \mathit{wait}, \mathit{start})$ variants over OW baselines (Table 3.2). For each test instance on each particular machine configuration, we simulate each variant $(\cdot, \cdot, \mathit{wait}, \mathit{start})$, 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 $(\cdot, \cdot, \mathit{wait}, \mathit{start})$, regardless of the sequencing method. Thus, a number in Table 3.2 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 $(\cdot, \cdot, \mathit{wait}, \mathit{start})$ the average response latencies are reduced at least by the factor of two.

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 online, existing jobs might be starved: these policies would additionally need to consider fairness.

Finally, while composition in FaaS model 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.

Chapter 4

Semi-Flexible Cloud Resource Allocation

In Function as a Service (FaaS) model, incoming load (events triggering particular function calls) is scheduled onto function execution environments running on worker machines. Cloud resource manager has to balance global efficiency — packing the machines as densely as possible — and serving quality — ensuring that individual machines are rarely, if ever, overloaded. This problem can be generalized to a wide range of cloud applications and cloud resource managers (like Borg [1], [71] and Resource Central [90]). In a general scenario, a typical cloud application uses two additional layers: (1) a (horizontal) autoscaler (e.g., [51]) that adds or removes instances in response to long-term changes in the application load; and (2) a load balancer (e.g., [52]) that dynamically assigns end-user queries to instances for a shorter-term balance. These two layers typically issue requests to, rather than fully coordinate with, the resource manager.

In this chapter, we show that in such a general scenario — by integrating the autoscaler, the resource manager (scheduler) and the load balancer — the cloud resources may be used more efficiently. In particular, we coordinate setting the number of instances (autoscaling), their placement on machines (scheduling) and the allocation of load to individual instances. Unlike the standard load balancing, our proposed method balances the load taking into account all the machine's assigned instances (and thus, all the applications).

The contemporary cloud software stack has enormous engineering complexity. Thus, to show the impact of our ideas, instead of a system study which would most probably be infeasible, we take a formal, algorithmic approach based on classic scheduling (which we additionally complement with simulations). This formal approach enables us to show trends and qualitative differences, but it requires some reasonable modeling of the problem.

As in Google's Borg [1], we focus on two key resources: the operational memory (RAM) and the computational power (CPU). We model a system hosting multiple *applications*, each processing a certain *load*. This load is distributed between the application's *instances*. For example, in Function as a Service (FaaS), a single function corresponds to our application. The set of end-user-driven invocations of this function (a stream of HTTP requests) is the application's load. Any invocation can be processed by any machine that has initialized this function's environment. As another example, our application corresponds to a single serving job in Google's Borg model [1], [71]; its instances correspond to the job's tasks; and a cluster-level load balancer assigns queries to these tasks. Finally, to derive tangible formal results, we assume that the memory requirement of an instance does not depend on the load processed by this instance. While we have no data to back up this assumption, various application classes should behave according to that model, with the memory requirements dominated by the software stack (libraries, etc.), pre-loaded datasets, or static data structures. We stress that this is not a core assumption (as it could be easily extended to, e.g., a linear function) but rather a standard modeling step that allows us to demonstrate qualitative results.

In this model, we analyze two natural combinatorial optimization problems: (1) bin-packing-like minimization of the number of used machines; (2) makespan-like minimization of the maximum load of any machine. Bin-packing models applications' requirements as hard constraints. This corresponds to, e.g. packing high-priority jobs by their limits in Borg; or, in an IaaS provider, packing VMs by their sizes (as requested by customers) and maintaining strict SLOs with no overcommitment. In contrast, the makespan-like approach explores the different nature of these resources. A memory requirement of an application cannot be (easily) compressed or throttled (cloud providers do not swap memory to disk [71]). Unlike memory, the CPU is compressible — it can be dynamically throttled. Of course, when throttled, the application slows down, which is tolerable for batch, while it should be avoided for serving applications. Thus, the makespan-like approach minimizes the load of the maximally-loaded machine, corresponding to the minimization of the maximal throttling. (The "makespan" is a metaphor: we assume that applications are executed concurrently.)

This chapter is structured as follows:

- We present a combinatorial optimization model of cloud application allocation with load balancing and fixed memory requirements. We formulate two general optimization problems: packing and balancing (Section 4.1).

- We prove NP-Hardness for both models in the general case. We also show optimal polynomial algorithms for equal requirements. (Section 4.2–4.3).

- We propose heuristics for the packing problem that take into account applications' semi-flexibility (Section 4.4).

- We simulate heuristics with instances based on Azure Public Dataset V2 [90] (Section 4.5).
- We present work related to semi-flexible cloud resource allocation (Section 4.6).
- We summarize our work and discuss results (Section 4.7).

4.1 A scheduling model of semi-flexible allocation

In this section, we formally define the optimization problem of cloud application allocation with memory requirements and CPU load balancing as a general integer linear program (ILP). We follow the classic scheduling notation [48] and extend notation presented in 2.1.

Let us be given m identical machines, each having a memory capacity of $Q \in \mathbb{Z}_+$ (measured in, e.g., bytes) and the CPU capacity of $P \in \mathbb{Z}_+$ (measured in, e.g., vCPUs or Borg's Normalized CPUs [1], [71]). We assume the machines are identical as cloud providers usually manage a few large groups of homogeneous machines (e.g., 4 machine types cover 98% of machines of a Google's over 12,000-machine cluster [91] — we thus solve a separate instance for each of these 4 large groups). Similarly, if a system uses VMs rented from an IaaS provider, it is natural to use a Managed Instance Group that requires all VMs to have the same instance type.

Let us also be given n *applications*. Contemporary cloud applications are usually horizontally-scalable: multiple *instances* of the same application, placed on multiple machines, jointly process the load of the application (e.g., for serving applications, each instance processes a fraction of the stream of incoming requests). In cloud, this mechanism is additionally used to increase reliability (e.g.: 3 or 5 always-on instances). While such lower bounds on the number of active instances can be easily incorporated into our approach, they are mostly orthogonal to our results (so we do not discuss them further).

There is a cost of maintaining multiple instances, however: each instance of the i-th application has its own integer memory demand of $0 < q_i \leq Q$ (measured in the same unit as the memory capacity). For example, if the i-th application is placed on two machines, it uses a total of $2q_i$ units of memory: q_i units on the first and q_i units on the second machine. We assume that, for the i-th application, q_i is constant — in particular, unrelated to the load assigned to the instance. This corresponds to memory requirements dominated by the software stack (libraries etc.), or the dataset, rather than dynamically changing with the processing load. Our model can be extended to memory requirement being a (perhaps linear) function of the load assigned, but we prefer to keep our model simple and the memory requirements constant in order to prove formal results and show qualitative conclusions.

One of our goals is to illustrate how much we can improve the utilization of the whole cluster by maintaining multiple instances. We will test our *multi-instanced models* against the standard scheduling models, later called *single-instanced.*

Moreover, we know p_i, the total load that needs to be processed by application i. The load $p_i > 0$ is expressed as the number vCPUs it requires (we use the same metric for the load and the capacity following Borg [1], [71]). In the single-instanced model, p_i units of the vCPU capacity on a single machine need to be reserved for the sole instance. In the multi-instanced model, the load balancer freely divides p_i between the application's instances as long as the reserved CPU capacity sums up to at least p_i. The amount of computation assigned to a particular instance may be fractional, too. For many serving applications, the load consists of a huge number of relatively tiny requests (single API calls or FaaS invocations). If the total QPS is in thousands, load balancer decisions can be reasonably approximated by fractional assignments.

We assume a classic, off-line and clairvoyant model with p_i and q_i known in advance, a common approach in cloud resource management research. The load p_i does not correspond to processing time, but to the total load of the i-th application. When customers deploy their applications in cloud, they are commonly required to upper-bound the total number of vCPUs and memory — and their application is then allocated based on these given upper-bounds. Additionally, serving applications are usually long-running: steady-state vCPU and memory requirements can be precisely estimated based on relatively simple models using historical trends [51].

Our aim is to assign applications to machines in such a way that all the incoming requests can be processed and that memory used on each machine does not exceed its capacity. As—in this model—it makes no sense to place two instances of the same application on the same machine (if such two instances were merged, they would process the same load using half the memory), we will use a 0-1 variable x_{ij} to determine whether the instance of the i-th application is placed on the j-th machine, or not. If positive, the p_{ij} variable will determine the total vCPU capacity reserved for the i-th application on the j-th machine. Furthermore, p_{ij}/p_i corresponds to the share of the whole traffic of the i-th application routed by a load balancer to the j-th machine. We thus always want the following constraints to be satisfied:

- As we do not overcommit memory on any machine, the memory utilization of all the instances placed on the j-th machine does not exceed machine's capacity Q, i.e.

$$\sum_i x_{ij} q_i \leq Q, \quad \text{for each } j; \tag{4.1}$$

- The total vCPU capacity reserved for the i-th application is greater or equal to the application's load p_i, i.e.

$$\sum_j x_{ij} p_{ij} \geq p_i, \quad \text{for each } i; \tag{4.2}$$

- Assignments x_{ij} are binary, i.e.

$$x_{ij} \in \{0, 1\}, \quad \text{for each pair of } i \text{ and } j. \tag{4.3}$$

In Figure 4.1, we present an example allocation of three applications, $\{\bigstar, \blacklozenge, \blacktriangledown\}$, to one machine. Total memory used by these three instances is equal to $q_\bigstar + q_\blacklozenge + q_\blacktriangledown \leq Q$, so the constraint (4.1) is not violated on this machine. At the same time, the total CPU used slightly exceeds the value of P, i.e., $p_{\bigstar j} + p_{\blacklozenge j} + p_{\blacktriangledown j} > P$ (so far we have not introduced machine-level constraints on the vCPU load). In fact, applications $\bigstar$ and $\blacktriangledown$ are assigned as much vCPU capacity as they require, i.e., $p_{\bigstar j} = p_\bigstar$ and $p_{\blacktriangledown j} = p_\blacktriangledown$. However, the vCPU capacity assigned to application $\blacklozenge$ is strictly less than its whole demand, i.e., $p_{\blacklozenge j} < p_\blacklozenge$. Thus, another instance of application $\blacklozenge$ has to be allocated on at least one other machine, as otherwise constraint (4.2) will not be satisfied.

We consider two natural optimization objectives. First, given a fixed pool of homogenous machines, we assign applications and workloads to machines in such a way that the maximum vCPU usage among all the machines is minimized:

$$\min \max_j \sum_i x_{ij} p_{ij}. \tag{4.4}$$

This objective models a cloud resource manager such as a single BorgPrime ([1], [71]) that allocates load on a single cluster of physical machines.

Second, given a set of applications and a constraint on the maximum vCPU usage on each machine, we minimize the number of used machines. This models a large customer minimizing the number of rented VMs while maintaining applications' SLOs. As formalization of this objective needs additional notation, we defer it to Sec. 4.3.

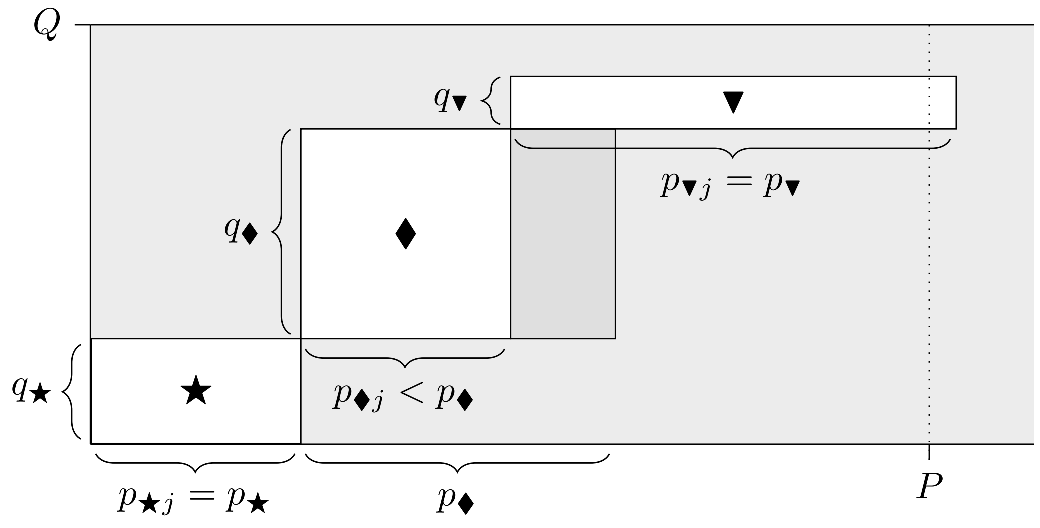


Figure 4.1: An example assignment of three applications to a single machine j in the multi-instanced model. Memory capacity Q (solid horizontal line) cannot be exceeded due to constraint (4.1), while vCPU capacity P (dotted vertical line) can.

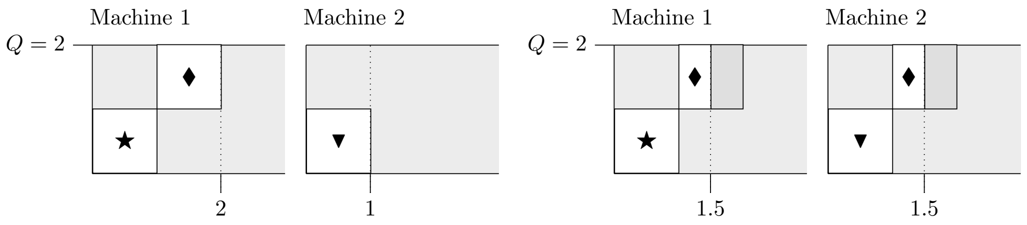
4.2 Optimal Balancing: Minimizing the Maximum CPU usage

In the optimal balancing problem, $\min \max_j \sum_i x_{ij} p_{ij}$, we start by analyzing polynomially-solvable special cases of common CPU and memory requirements; we then proceed to prove NP-hardness for arbitrary CPU requirements (with unit memory requirements); and arbitrary memory requirements (with unit CPU requirements).

4.2.1 Common CPU and memory requirements

We now assume that all applications have the same CPU requirement of p and the same memory requirement of q. Although it would seem that it makes no sense to have multiple instances of any application, especially if additionally $p = 1$, it is not true, as the following example shows.

Example 1. Let us consider a two-machine environment ($m = 2$) where $Q = 2$, and three different applications such that $p_i = q_i = 1$ for $i \in \{\bigstar, \blacklozenge, \blacktriangledown\}$. If one assigns applications $\bigstar$ and $\blacklozenge$ to the first machine and application $\blacktriangledown$ to the second machine, then $\max_j \sum_i x_{ij} p_{ij} = 2$ (the maximum is reached on the first machine, see Figure 4.2(a)). However, if one assigns application $\bigstar$



(a) All the applications have a single instance (b) Application 2 has two instances

Figure 4.2: Example assignment of three applications to two machines

to the first machine, application ▼ to the second machine, and application ♦ to *both* the first and the second machine in such a way that $p_{\bigstar 1} = p_{\bigstar 2} = 0.5$, then $\max_j \sum_i x_{ij} p_{ij} = 1.5$ (see Figure 4.2(b)) which is the lower bound on the maximum CPU usage (as $\sum_i p_i/m = 1.5$).

Algorithm 2 shows how to solve this problem for any q and Q. To simplify its description, we assume now that $q = 1$ (it can be done without loss of generality). Notice that $m \cdot Q \geq n \cdot q = n$ must hold (at least one instance of each application must be placed on some machine). Otherwise, there would exist no feasible assignment of applications to machines. As m, n and Q are integers, it must also hold that $Q \geq \lceil n/m \rceil$. In this special case we assume that each application requires a total of p CPU units. If $n \bmod m = 0$, then the optimal solution can be obtained by assigning any n/m applications to each of the machines, without adding multiple instances. This solution leads to the optimal value of $\max_j \sum_i x_{ij} p_{ij} = pn/m$. Now, assume that $n \bmod m > 0$ (which entails $\lceil n/m \rceil > n/m > \lfloor n/m \rfloor$). If $Q > \lceil n/m \rceil$, then at least $\lfloor n/m \rfloor + 2$ instances can be placed on each machine. If so, the optimal solution with $\max_j \sum_i x_{ij} p_{ij} = pn/m$ can be obtained with the McNaughton's algorithm [92]. However, if $Q = \lceil n/m \rceil$, the things get a little complicated, as a single machine can be assigned at most $\lfloor n/m \rfloor + 1$ instances. The actual maximum CPU usage strongly depends on the value of $n \bmod m$, which we assumed is strictly greater than 0. For example, if $n \bmod m > m/2$, then there exists an optimal schedule in which at least one machine will be assigned $\lceil n/m \rceil$ instances, none of which will be duplicated on any other machine. Thus, the optimal value of $\max_j \sum_i x_{ij} p_{ij}$ must be equal to $p \cdot \lceil n/m \rceil = p \cdot (\lfloor n/m \rfloor + 1)$. On the other hand, if $n \bmod m = 1$, then this single additional instance can be placed on each machine and the load can be evenly divided between all its instances leading to $\max_j \sum_i x_{ij} p_{ij} = p \cdot (\lfloor n/m \rfloor + 1/m)$. In general, if $Q = \lceil n/m \rceil$ and $n \bmod m > 0$, then the optimal value of $p \cdot (\lfloor n/m \rfloor + r)$ can be reached where $r = 1/\lfloor m/(n \bmod m) \rfloor$.

The latter results require some justification. Let us observe that in the considered case ($Q = \lceil n/m \rceil$ and $n \bmod m > 0$) there always exists an optimal assignment in which each machine hosts at least $\lfloor n/m \rfloor$ single-instances applications and *at most* one of the remaining $n \bmod m$. Thus, let $n' = n \bmod m$, which implies $0 < n' < m$. The exact value of r can be derived by answering the following question: *what is the optimal CPU usage for m machines and n' applications, if each machine can host only one application?* If n' divides m, then one can split each application into exactly m/n' identical instances, one per machine. Optimally, every instance would then process exactly $1/(m/n')$ of the load p. In general, it may be not possible to split each application into exactly m/n' identical instances. As we want to maximize the minimum number of instances, in the optimal assignment some applications will have $\lfloor m/n' \rfloor$ instances, while some (zero, if n' divides m) will have $\lfloor m/n' \rfloor + 1$ instances. Thus, the minimal achievable fraction of load processed by a single instance will be $1/\lfloor m/(n \bmod m) \rfloor$ of p, as the fewer instances, the more load each of them needs to process.

4.2.2 Arbitrary CPU or memory requirements

If either CPU or memory requirements are arbitrary (i.e., application-dependent), then the problem becomes NP-Hard.

Lemma 1. Minimizing the maximum CPU usage is NP-Hard:

1. for unit CPU requirements ($\forall i : p_i = 1$) and arbitrary memory requirements;

2. for unit memory requirements ($\forall i : q_i = 1$) and arbitrary CPU requirements.

Proof. We show that 3-Partition (where we need to split a set of $3m$ values into m threes in such a way that the sum of each three is the same) reduces to these special cases. Let us be given a set of $3m$ positive integers $a_1, a_2, \ldots, a_{3m}$ (with their sum being a multiple of m). We create an instance of the balancing problem with $n = 3m$ applications such that, depending on the case:

1. $p_i = 1$ and $q_i = a_i$,

2. $q_i = 1$ and $p_i = a_i$,

for each $i \in \{1, 2, \ldots, 3m\}$. We are also given m machines such that, depending on the case:

Algorithm 2 Constructing the assignment that guarantees the minimal maximum vCPU usage if $p_i = p$ and $q_i = q$. For a fixed value of m, this algorithm works in $O(n)$.

```
$Q \leftarrow q \cdot (Q \div q)$                          ▷ Make $Q$ a multiple of $q$
$\mathcal{A} \leftarrow$ a stack of all the applications
        ▷ Each application requires a total of $p$ CPU units and $q$ memory.
for $j \leftarrow 1, 2, \ldots, m$ do
    for $k \leftarrow 1, 2, \ldots, \lfloor n/m \rfloor$ do
        $J \leftarrow$ Pop($\mathcal{A}$)
        Assign $p$ units of vCPU load of application $J$
          to the $j$-th machine
$r \leftarrow 0$
if $|\mathcal{A}| > 0$ then                        ▷ There are still applications left
    if $Q > q \cdot \lceil n/m \rceil$ then                 ▷ Use McNaughton's algorithm
        $r \leftarrow p \cdot (n/m - \lfloor n/m \rfloor)$                ▷ Available machine load
        $i \leftarrow 1$                            ▷ Index of the considered machine
        $rr \leftarrow r$                           ▷ Disposable load on the machine
        while $i < m$ do
            $J \leftarrow$ Pop($\mathcal{A}$)
            Assign $rr$ units of CPU load of
              application $J$ to the $i$-th machine
            $i \leftarrow i + 1$                        ▷ Move to the next machine
            Assign $p - rr$ units of CPU load of
              application $J$ to the $i$-th machine
            $rr \leftarrow r - (p - rr)$
    else                                    ▷ Assign applications unevenly
        $r \leftarrow p / \lfloor m/(n \bmod m) \rfloor$               ▷ Available machine load
        $i \leftarrow 1$                            ▷ Index of the considered machine
        $rp \leftarrow 0$                          ▷ Disposable load of an application
        while $i \leq m$ do
            if $rp = 0$ then
                if Empty($\mathcal{A}$) then
                    end while
                $J \leftarrow$ Pop($\mathcal{A}$)
                $rp \leftarrow p$
            Assign $\min\{rp, r\}$ units of CPU load of
              application $J$ to the $i$-th machine
            $rp \leftarrow r - \min\{rp, r\}$                    ▷ Load left for app. $J$
            $i \leftarrow i + 1$
```

1. $Q = \frac{1}{m}\sum_{i=1}^{3m} q_i$,
2. $Q = 3$.

The question is, depending on the case, *does there exist an assignment of instances to machines such that*:

1. $\max_j \sum_i x_{ij}p_{ij} \leq 3?$,
2. $\max_j \sum_i x_{ij}p_{ij} \leq \frac{1}{m}\sum_{i=1}^{3m} p_i?$.

Note that any 3-Partition instance is a yes-instance if and only if the corresponding instance of our problem is a yes-instance. As the transformation can be performed in polynomial time, the NP-Hardness follows. □

4.3 Minimizing the number of machines used

In the bin-packing variant of the problem, we minimize the number of used machines with an additional constraint on the maximum vCPU utilization on any machine, P. In the ILP formulation, we introduce an indicator binary variable $y_j \in \{0, 1\}$ that marks a machine that is used (that is assigned some load). We also add a constraint:

$$\sum_i x_{ij}p_{ij} \leq Py_j. \tag{4.5}$$

The number of machines (hence, the number of y_j variables) is upper bounded by $\sum_{i=1}^{n}\lceil p_i/P\rceil$, corresponding to an allocation in which each machine is assigned at most one instance of some application, with the vCPU capacity of P, until all the CPU requirements are met. The goal is thus:

$$\min \sum_j y_j.$$

We analyze special cases of: (1) unit; (2) common; and (3) arbitrary CPU or memory requirements.

4.3.1 Unit CPU and memory requirements

This special case is simple. Consider a machine for which the values of P and Q are given. Note that both the P and Q values are assumed to be integers. In a single-instanced model, exactly $\min\{P, Q\}$ applications are allocated on each machine j. Surprisingly, having multiple instances does not decrease the number of used machines, as the following lemma shows.

Lemma 2. If $p_i = q_i = 1$ for each i, then there exists an optimal assignment of instances to machines where all applications have exactly one instance.

Proof. Let us consider any optimal assignment in which an application has more than one instance. For each machine j, let us denote by $\mathcal{I}_j$ the set of all applications i placed on that machine ($x_{ij} = 1$), but only with partial load ($p_{ij} < 1$). Thus, there must exist at least two machines, j and k, for which $|\mathcal{I}_j| > 0$ and $|\mathcal{I}_k| > 0$. In consequence, the set $\mathcal{I} = \bigcup_j \mathcal{I}_j$ is not empty. Notice that the total CPU usage of the applications in $\mathcal{I}$ is $|\mathcal{I}|$, and that the total memory usage of these applications is $\sum_j |\mathcal{I}_j| \geq 2|\mathcal{I}|$.

We now reassign applications to machines in such a way that the number of used machines does not increase, but all the applications have exactly one instance. Let all the applications from $\mathcal{I}$ be removed from all the machines. Now, consider any machine j that was affected by this operation. If one assigns to this machine as many complete ($p_{ij} = 1$) applications i from the $\mathcal{I}$ set as possible, the total CPU load on machine j will be not lower than the initial load. If this is so for all the machines, the lemma follows. Notice that the statement might be not true only if the number of applications left to be assigned is lower than the capacity of the selected machine. However, in such a case the lemma also follows, as the limit of P is not achieved. $\square$

Lemma 2 leads us to the following greedy approach. Determine the critical capacity $\min\{P, Q\}$ of a machine. Then, until there are no applications left to be assigned, place as many as possible complete ($p_{ij} = 1$) instances of applications on a new machine.

The above argument generalizes to any case in which $p \mid P$ and $q \mid Q$. Indeed, if $q \mid Q$ then the memory requirements can be scaled to $q = 1$ with Q changed to $Q \div q$ (the same argument holds for p).

4.3.2 Common CPU and memory requirements

As seen in the previous section, when $p \mid P$ and $q \mid Q$, there exists an optimal assignment of applications to machines such that no application uses more than one instance. If $q \nmid Q$, then without the loss of generality, we can reduce Q to $q \cdot (Q \div q)$ (as no application would fit into the remaining capacity $Q \bmod q$ anyhow). Notice that $q \mid q \cdot (Q \div q)$. In other words, we can always assume—without the loss of generality—that $q \mid Q$ and, equivalently, that $q = 1$.

However, if $p \nmid P$, then Lemma 2 is not true anymore, as the following example shows.

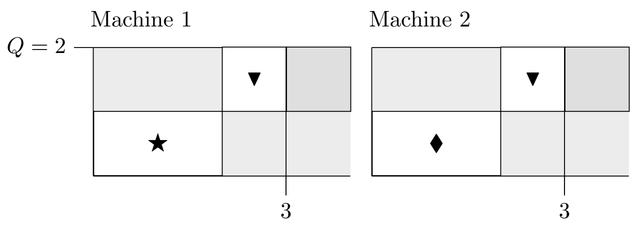


Figure 4.3: When ▼ has more instances, the allocation uses just two machines (while a single-instanced allocation uses three).

Example 2. Assume $Q = 2$ and $P = 3$, and that we are given three applications, $\{\bigstar, \blacklozenge, \blacktriangledown\}$, with $q_i = 1$ and $p_i = 2$ ($p \nmid P$). In the single-instanced model, the optimal solution allocates each application to a separate machine, with $m = 3$. A multi-instanced model allows us to split one of the applications as in Figure 4.3, and in consequence use just two machines.

Applications using multiple instances can reduce the number of used machines by the factor of almost two, as proved by the following lemma.

Lemma 3. Let m be the optimal number of machines for the single-instanced model; and let m' be the optimal number of machines for the multi-instanced model. (1) For any instance, $m/m' < 2$; (2) for any given real $\varepsilon > 0$, there exists an instance such that $2(1 - \varepsilon) < m/m' < 2$.

Proof. We first show by contradiction that for any instance $m/m' < 2$. Notice that $m \geq m'$ and that $m = 1$ if and only if $m' = 1$. Thus, assume that $m' > 1$, $m \geq 2m'$, and that the values of Q, P, q and p are arbitrary. Consider the single-instanced model (which requires $q \leq Q$ and $p \leq P$). Let L be the maximum CPU load among all the machines. As applications have common requirements, in any optimal solution, applications can be relocated in such a way that on the first $m-1$ machines the load is equal to L, and on the m-th machine the load L' is positive, yet not greater than L. Now, consider the multi-instanced model. As $m \geq 2m'$ and $m' > 1$, there must exist at least one machine for which the total CPU load H meets the condition $H \geq L+L'$.

This is so, as the average load on m' machines is equal to:

$$\begin{aligned}\frac{(m-1)L+L'}{m'} &\geq \frac{(m-1)L+L'}{m/2} = \\ &= \left(2-\frac{2}{m}\right)L+\frac{2}{m}L' = \\ &= L+\left(1-\frac{2}{m}\right)L+\frac{2}{m}L' \geq \\ &\geq L+\left(1-\frac{2}{m}\right)L'+\frac{2}{m}L' = \\ &= L+L'.\end{aligned}$$

However, by the above, the number of applications assigned to this machine is at least $(L+L')/p$. Thus, $Q \div q \geq (L+L')/p$ and $P \geq L+L'$. Consequently, in the single-instanced model, machines $m-1$ and m could be merged — which means m is not optimal, leading to a contradiction.

Now, let us show that, for any $\varepsilon > 0$, there exists an instance such that $2(1-\varepsilon) < m/m'$. Let $n > 2$ be the number of different applications, and let $q_i = 1$ for each $i \in \{1, 2, \dots, n\}$. Let also $Q = 3$, $p = n$ and $P = 2n-1$. If we consider a single-instanced model, the optimal number of machines, m, is equal to n. On the other hand, the total load processed by n applications is n^2 and each of the machines is capable of processing the load of size $2n-1$. As a consequence, the lower bound on the number of machines m' for the multi-instanced model is $\lceil n^2/(2n-1)\rceil$. An optimal assignment using exactly $\lceil n^2/(2n-1)\rceil$ machines, where at most three different applications are assigned to each machine, can be found by assigning jobs to machines greedily, one machine after another. Thus,

$$\begin{aligned}\frac{m}{m'} &= \frac{n}{\lceil n^2/(2n-1)\rceil} \\ &> \frac{n}{n^2/(2n-1)+1} = \frac{2n^2-n}{n^2+2n-1} \xrightarrow[n\to\infty]{} 2_-.\end{aligned}$$ $\square$

In order to find an optimal assignment in polynomial time, we observe that:

1. on any machine, we can allocate $\min\{P \div p, Q \div q\}$ whole applications;

2. it is suboptimal to allocate many instances of applications to a single machine if a single instance of the application could be assigned instead (cf. Lemma. 2);

3. if the optimal load ($\min \max_j \sum_i x_{ij} p_{ij}$) is known, we can compare it to P to see whether additional machines are necessary.

Thus, the general idea of the algorithm is as follows (Algorithm 3). Given the lower m_L and the upper m_U bounds on the possible number m of machines, we test—based on binary search procedure and Section 4.2.1—whether the optimal maximum CPU load exceeds P. Depending on the answer, we limit the range of m. Note that Algorithm 3 works for the case of $p \mid P$, too. Also note that, given the optimal number of machines m, one can assign applications to machines based on the Algorithm 2.

Algorithm 3 Finding the minimal number of machines if $p_i = p$ and $q_i = q$

```
$m_L \leftarrow \lceil \max\{n/(Q \div q), n \cdot p/P\} \rceil$    ▷ Lower bound on the number of machines
$m_U \leftarrow n \cdot \lceil p/P \rceil$
while $m_U > m_L$ do
    $m \leftarrow (m_L + m_U) \div 2$
    if $n \bmod m = 0$ or $Q \div q > \lceil n/m \rceil$ then
        $P' \leftarrow p \cdot n/m$
    else
        $P' \leftarrow p \cdot (\lfloor n/m \rfloor + 1/\lfloor m/(n \bmod m) \rfloor)$
    if $P' > P$ then
        $m_L \leftarrow m + 1$
    else
        $m_U \leftarrow m$
return $m_L$
```

4.3.3 Arbitrary CPU or memory requirements

Analogically to the load balancing problem (Section 4.2.2), if either CPU or memory requirements are arbitrary (i.e. application-dependent), then the problem becomes NP-Hard.

Lemma 4. Minimizing the number of machines used is NP-Hard:

1. for unit CPU requirements ($\forall i : p_i = 1$) and arbitrary memory requirements;
2. for unit memory requirements ($\forall i : q_i = 1$) and arbitrary CPU requirements.

Proof. We show that 3-Partition reduces to these special cases. Let us be given a set of $3m$ positive integers $a_1, a_2, \ldots, a_{3m}$ (with their sum being a multiple of m). We create an instance of our problem with $3m$ applications such that, depending on the case:

1. $p_i = 1$ and $q_i = a_i$,
2. $q_i = 1$ and $p_i = a_i$,

for each $i \in \{1, 2, \ldots, 3m\}$. We are also given m machines such that, depending on the case:

1. $Q = \frac{1}{m} \sum_{i=1}^{3m} q_i$ and $P = 3$,
2. $P = \frac{1}{m} \sum_{i=1}^{3m} p_i$ and $Q = 3$.

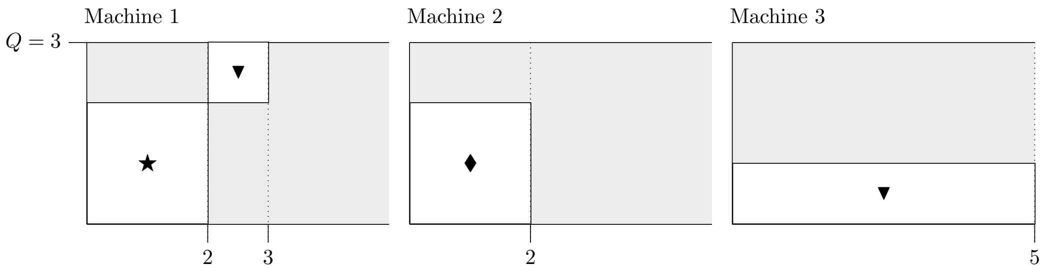
The question is: *does there exist an assignment of instances to machines such that* $\sum_j y_j \leq m$*?* Note that any 3-Partition instance is a yes-instance if and only if the corresponding instance of our problem is a yes-instance. As the transformation can be performed in polynomial time, the NP-Hardness follows. □

4.4 Heuristics

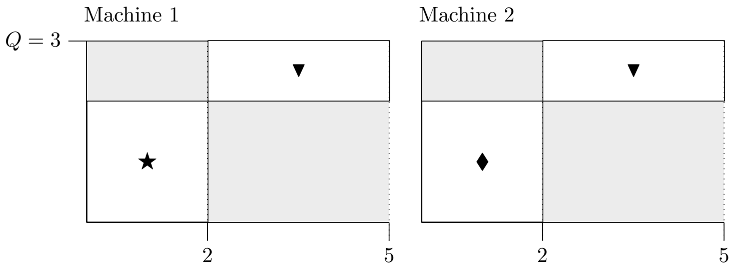
As with arbitrary requirements both the balancing and the bin-packing problems are NP-hard, in this section we propose a number of heuristics. We focus on bin-packing (minimizing the number of used machines), because this problem is perhaps more applicable of the two (as the machine's CPU capacity should not be exceeded in the steady state). We start with baselines: heuristics packing single-instanced applications. Then, we extend them for the multi-instanced model.

A cloud cluster usually consists of large groups of similar machines. For each of these machines, the CPU capacity and memory size is known in advance. Moreover, usage limits can be set on these resources. For example, on a single machine, one may not want to exceed 95% of memory capacity and 80% of the total CPU capacity. These thresholds directly translate to values of P and Q.

Before we introduce our heuristics, we discuss the limits on the values of p_i and q_i. Let us consider an i-th application. It must hold that $q_i \leq Q$ as otherwise an instance of this application could not be placed on any machine. However, our baselines for single-instanced model force us to assume that for each i-th application $p_i \leq P$. In general, one could assume that if there exists



(a) The ▼ application is forced to use a dedicated Machine 3



(b) Optimal assignment

Figure 4.4: Example assignment of three applications to two machines. On the top, we assume that an application for which $p_i > P$ is assigned a dedicated machine. On the bottom, we present an optimal assignment.

an application for which $p_i > P$, then this application could be assigned — in advance — to $p_i \div P$ dedicated machines. Then, the remaining $p_i \bmod P$ units of load could be assigned to some other machine based on the considered heuristic. This approach may not lead, in general, to optimal solutions, as Example 3 shows. Thus, we cannot introduce it in our baselines.

Example 3. Let us consider a multi-machine environment where $P = 5$, $Q = 3$, and three different applications, $\{★, ♦, ▼\}$, such that $p_★ = p_♦ = q_★ = q_♦ = 2$, $p_▼ = 6$ and $q_▼ = 1$. If P units of application's ▼ load will be assigned to a dedicated machine, then the total number of machines required to process all the load will be equal to 3 (Figure 4.4(a)) while the optimal number is equal to 2 (Figure 4.4(b)).

4.4.1 Baselines for the single-instanced model

As the baselines, we use standard list heuristics parametrized on two levels: (1) for a specific application, how to choose the machine; and (2) the order in which the applications are processed. We consider the following rules:

First-Fit Find the first machine on which the application i fits (the machine's free CPU is at least p_i; and the machine's free memory is at least q_i).

Next-Fit Find the next machine on which the application i fits (in a cyclic way: after reaching the last opened machine, we start with the first one).

Worst-Fit Find the machine for which, after the application i is placed, the remaining free memory is largest. Each of these rules opens a new machine if the application does not fit on any machine. Also note that we use worst-fit, rather than best-fit, in order to leave as much free memory on a machine as possible for the coming applications.

We consider the following application orders:

Mem-Increasing / Mem-Decreasing Choose the application with the largest/lowest memory requirement.

CPU-Increasing / CPU-Decreasing Choose the application with the largest/lowest CPU requirement.

Ratio-Increasing / Ratio-Decreasing Choose the application with the largest/lowest value of p_i/q_i, i.e., the ratio of CPU and memory requirements.

Random Choose applications in random order.

4.4.2 Heuristics for the multi-instanced model

Our heuristics are based on the binary search (Algorithm 3). We explore the range of possible numbers of machines $[m_L, m_U]$. For a tested m, we apply one of the heuristics below and check whether the assignment does not exceed P on any used machine. We consider the following heuristics:

CPU-Oriented This heuristic is based on the following assumption: *we want to add instances to applications that have the highest CPU requirements.* The whole applications are placed on machines based on the First-Fit/Next-Fit/Worst-Fit rule, starting from the applications with the lowest CPU requirements. It may happen that at some point none of the remaining applications will fit any of the machines without adding instances. Then, one application after another, starting from the one with the largest memory requirements, we greedily fill the machines with as large instances as possible. We do it based on the First-Fit/Next-Fit/Worst-Fit rule, and taking into account the limits of P and Q.

Mem-Oriented This heuristic is based on the following assumption: *we want to add instances to applications that have the lowest memory requirements.* The whole applications are greedily placed on machines based on the First-Fit/Next-Fit/Worst-Fit rule, starting from the applications with the largest memory requirements. It may happen that at some point none of the remaining applications will fit any of the machines without adding instances. Then, one application after another, we greedily fill the machines with as large instances as possible. We do it based on the same rule, and taking into account the limits of P and Q.

4.5 Experiments

Although the theoretical results of Lemma 3 are promising, they do not take into account the peculiarities of load processed in clouds. For this reason, we evaluate our heuristics using instances generated from the Azure Public Dataset V2 (VM Trace).

4.5.1 Data preprocessing

The Azure Public Dataset V2 (VM Trace) provides information about Azure VM workload collected over 30 consecutive days in 2019. The requirements of each VM are bucketed by their memory usage (0-2 GB, 2-4 GB, 4-8 GB, 8-32 GB, 32-64 GB or more than 64 GB) and core count (0-2, 2-4, 4-8, 8-12, 12-24, and more than 24). The VMs in the trace are grouped into *deployments* — sets of virtual machines deployed and managed together by a single client. We map each deployment to a separate application. We also filter out 513 out of 16,977 deployments which use buckets with indefinite upper bounds.

We map q_i to the maximum upper end-point of the memory buckets assigned to all VMs from the deployment. In 72% of the deployments, all VMs are in the same memory buckets; and in further 14%, they belong to exactly two buckets.

To derive p_i, we sum the average (fractional) usage of the vCPU cores over VMs in the deployment. As the trace defines only the range of vCPUs used by a VM, we use the upper end of the VM's CPU bucket. For example, if a VM is assigned a bucket of 8-12 cores, and its average CPU usage is 0.6, then we map that to $12 \cdot 0.6 = 7.2$ virtual cores.

We thus use the application's memory *limits* (upper bounds) for memory requirements, but the application's CPU actual *usage* for the load. This is so as the CPU — a compressible resource — is easier to vertically-scale without much disruption to the running VM, so the CPU limit should be closer to the (perhaps high percentile of) CPU usage.

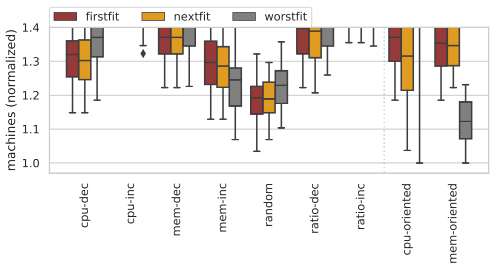
In our experiments, we use machines with $P = 32$ virtual cores and varying capacity Q of RAM: 64, 96, 128, 256, 512 and 1024 GB (we use Q as a parameter of the experiment). In an appendix [93], we present results for additional configurations. We start with 64 GB, as 99.6% of VMs in the trace have at most 64GB of RAM. We stop at 1024 GB, because, as we later show, memory ceases to be a critical resource from roughly this value (thus, a rational cloud provider would not have machines larger than this size). We also restrict instances to applications with the total vCPU requirement of at least 1 (VMs that use less than 1 vCPUs share it with other VMs which impacts the quality of service) and at most 32 (otherwise, in configurations with 64 GB machines, some machines would host just a single VM, regardless of the algorithm used).

After this mapping, in the 16,464 applications the median p_i is 5.0 and the mean is 8.4; the median q_i is 8 GB and the mean q_i is 14 GB. Thus, a 32-core machine accepts roughly 6 "median" applications when considering only the CPU requirements; and between 6 (for 64 GB of RAM) and 128 (for 1024 GB of RAM) "median" applications when considering only the memory requirements.

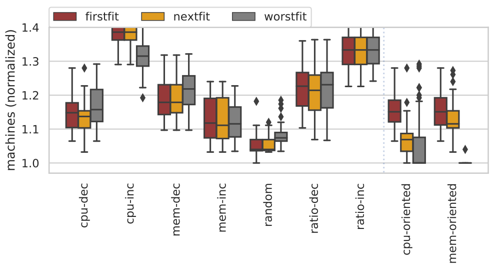
We generate 50 instances. Each instance has 100 applications selected randomly from the base set (without replacement). We simulate each heuristic on each instance and each machine configuration. We measure the number of used machines which directly corresponds with the average utilization (the lower the number of machines, the higher the utilization). To meaningfully compare results between different instances that can have different loads, we normalize the number of machines by the classic lower bound of the average load (in our CPU/memory case, the bound is extended by the average memory requirement): $\max\left(\lceil \sum_i p_i/P \rceil, \lceil \sum_i q_i/Q \rceil\right)$. Figure 4.5 shows the results.

4.5.2 Results

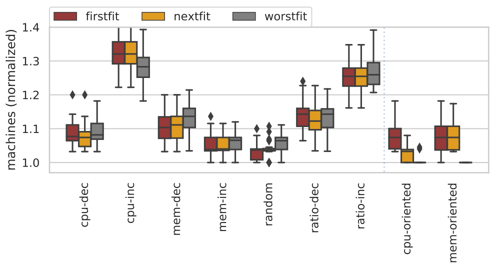
With enough memory (512-1024 GB, see Figure 4.5(ef)), our heuristics generate results equal to the lower bounds. The reason is that for large amounts of memory, memory is no longer a scarce commodity, thus the cost of maintaining multiple instances is negligible. Thus, one can easily obtain the lower-bound of $\lceil \sum_i p_i/P \rceil$. This holds regardless of the algorithm used, which



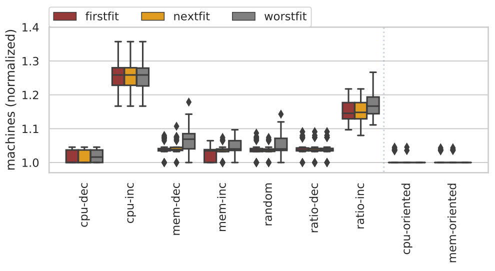
(a) 64 GB



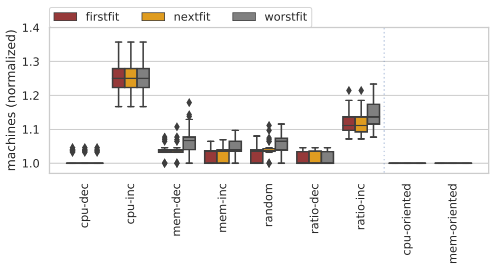
(b) 96 GB



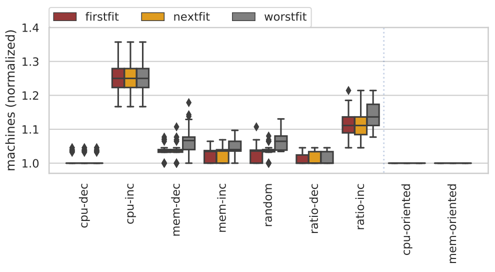
(c) 128 GB



(d) 256 GB



(e) 512 GB



(f) 1024 GB

Figure 4.5: Comparison of performance of the algorithms for machines with 32 vCPUs and varying amount of memory. Each boxplot shows statistics over 50 individual experiments. The line inside the box corresponds to the median and the height of the box indicates the first and the third quartile. The whiskers extend to the most extreme data point within 1.5 $\times$ IQR (inter-quartile range).

shows that simply having multiple instances is more important than how precisely we create them. On the other hand, baseline heuristics perform similarly, as the machines can be almost fully utilized when no application is split.

When the amount of memory is low (64 GB, see Figure 4.5(a)), adding instances is costly. Moreover, the number of instances that may be maintained by a single machine is small. Thus, algorithm performance strongly depends on the choice of the applications that have multiple instances. In Figure 4.5(a), both single-instanced and mutli-instanced heuristics perform poorly (at least 10% worse) compared to an optimistic lower-bound. However, the differences between methods are notable. The MEM-ORIENTED+WORST-FIT rule provides the best results (roughly 10% over the lower bound).

In the intermediate configurations (96-256 GB, see Figure 4.5(bcd)), multi-instanced heuristics show a significant improvement over single-instanced approaches, often leading to solutions achieving the lower bound. However, it still matters which applications have multiple instances: while in the case of 96 GB or RAM, MEM-ORIENTED+WORST-FIT is optimal, other heuristics are not — and the differences diminish with the increased memory capacity.

4.6 Related Work

Below, we review related work in combinatorial optimization and in VM placement. We refer to [94] for a survey on a broader topic of VM placement in the cloud datacenter.

The divisible load [50] scheduling model is related to our approach: the difference is that the divisible load does not consider the memory requirements.

When single-instanced applications of size q_i need to be placed on infinitely-efficient machines of capacity Q in order to minimize the number of machines used, our problem reduces to BIN-PACKING. Some heuristics solving optimally more than 95% of analyzed instances of the BIN-PACKING problem are known [95]. The key difference between our approach and the multi-dimensional bin-packing [40] is that in multi-dimensional bin-packing all requirements are fixed.

A related problem is also the one-dimensional FRACTIONAL BIN-PACKING: objects can be split across multiple bins (which is a linear programming relaxation of the BIN-PACKING problem). Variants in which the share of each object assigned to a single bin must be the same [96], or in which packing together two or more items make them use less resources than the sum of their individual requirements [97] are considered.

We consider the more general case of two-dimensional bin-packing in the multi-instanced model for which one dimension is always constant while the other one can change. Thus, we now focus on selected results that are related to these characteristics. In [98] bin-packing problem and the multiprocessor scheduling problem are connected: they minimize the number of workers or days required to produce certain amounts of goods. [99] analyzes how virtual machines with dynamic workload can be managed when the amount of resources required by them changes in time. They extend the typical load balancing with live migration to keep all the virtual machines in a limited number of active nodes. [100] also studies VM placement with live migrations.

VM placement with bin-packing (and its variants) are considered in [101]–[109]. Specifically, [108] shows improved approximation factors when load prediction is available. [107] analyzes heuristics for bin-packing taking into account the power-efficiency of the host. [109] analyzes VM placement with memory sharing (e.g. common libraries).

[110]–[114] are closest to our results as they map to special cases of our model. [110], [111] assume $q_i = 1$ and find any feasible assignment. [112] shows a (3/2)-approximation algorithm for $q_i = 1$ and $Q = 2$; and (7/5)-approximation for an arbitrary Q; [113] shows a PTAS for each of these cases. [114] analyzes $q_i = 1$ and arbitrary p_i on perhaps-failing machines (with high probability the assignment must fulfill the demand of each application). In contrast, our theoretical results solve optimally in polynomial time a special case with $q_i = q$ and $p_i = p$, i.e., equal memory and processing requirements.

4.7 Summary and discussion

We study a two-dimensional resource management problem with applications having multiple instances. While instances of an application have the same memory requirements, the CPU load can be freely balanced between them. From systems perspective, this approach integrates the scheduler, the autoscaler and the load balancer.

We present a number of theoretical results. We consider two related objectives: (1) minimization of the maximum load processed by a single machine; (2) minimization of the number of machines used. We demonstrate that both are NP-Hard in general, even when one of the dimensions of the problem is unit-sized. We also show polynomial algorithms that solve special cases with applications having equal requirements. We also provide strong theoretical motivation for having multiple instances: when bin-packing, replication may reduce the number of machines by a tight factor of $2 - \varepsilon$.

For the general case of the bin-packing problem, we propose heuristics. We simulate them on instances derived from the Azure Public Dataset. In the intermediate cases of 96-256 GB of RAM, compared with various single-instanced baselines, our heuristics reduce the number of used machines often achieving the lower bound.

Chapter 5

Scheduling methods for independent calls on single node

When an *end-user* invokes a function, this *invocation* is processed on the infrastructure managed by the FaaS provider. The cluster load balancer dispatches the incoming requests to worker nodes. As the worker can process multiple requests simultaneously, numerous ways exist to assign them the required resources. In this chapter, we analyze node-level scheduling methods which take advantage of characteristics of FaaS workload.

The FaaS system has to operate with varying loads. A recent study of the Azure Functions trace [75], a major public provider of FaaS, shows that the rate of requests is uneven, with peaks of short duration. Even though these peak loads are short (minutes rather than hours), the performance of the service under peak load drives resource allocation in the steady state, because end-user serving workloads are often optimized for the tail (95th or 99th percentile) response latency [115]. Adding a new worker node to a FaaS cluster (horizontal autoscaling) does not address this problem, as it takes at least dozens of seconds [116]. Thus, currently, the only way to handle peak loads without compromising the tail latency is to heavily over-provision: to run the services at low average CPU utilization (20%-50%), so that any peak can be handled gracefully on nodes. We address this issue by proposing algorithms that improve performance when the node load is high.

In FaaS, each function can be invoked numerous times, e.g. in response to repeated HTTP requests coming from various end-users. Thus, the local (node) scheduler can make online decisions on how to assign these invocations to available CPU cores based on invocations from the past. The information used may include, among others, the frequency of invocations and their observed past execution times. For this reason, theoretical lower bounds for the competitiveness of online strategies such as SPT or SRPT (see, e.g., [117]–[119]) can be too conservative.

These local scheduling decisions can be made implicitly by the kernel scheduler (at the operating system level). However, the kernel has a low-level perspective on the scheduling problem. In particular, the kernel is not aware of how individual FaaS invocations map to threads, so, in the context of FaaS, it cannot make dynamic, function-related decisions. As a consequence, the kernel is forced to using variants of the round-robin approach, where individual invocations are processed in turns and thus repeatedly preempted.

In this chapter, we consider optimizations that can be done on node-level. We consider a set of functions that have already been loaded into the memory of a single node in a large cluster. We intentionally omit the process of the function-to-node assignment to show that the performance of the whole cluster can be improved on the node-level too. Such an improvement is orthogonal to improvements in function placement [18], [88], load-balancing [88] or auto-scaling of the clusters [120],

We state that in practice one rarely deals with extreme generality (as in the theoretical, worst-case results) and that better decisions can be made by taking into account information readily available on a local FaaS node. This is so independent of whether the information is known a priori or is guessed (predicted) based on historical data. We show that the overall performance of the system can be increased by implementing other reasonable heuristics on the local scheduler level with no significant computational or memory cost—as long as we can use the information about how the individual invocations link to the functions. We validate this claim with computational experiments using the real-life data recently published as the Azure Functions Trace [75].

This chapter is structured as follows:

- We define a theoretical model of node-level scheduling for FaaS (Section 5.1). We adapt the real-life data from the Azure Function Trace to reflect our model (Section 5.2–5.3).
- We propose a number of theoretically-grounded heuristics and a new one, *Fair Choice*, that can be used by the local scheduler to make decisions online. We also show that these heuristics can be implemented without a significant increase of auxiliary computations (Section 5.4).
- By simulations, we show that applying heuristics based on past information leads to a reduction in latency-related objectives, compared to the preemptive round-robin (corresponding to standard scheduling used by the operating system, Section 5.5).
- We present work related to node-level scheduling opimisations (Section 5.6).
- We summarize our work and discuss results (Section 5.7).

In Chapter 6 we present our implementation of methods introduced in this chapter in OpenWhisk, an open source FaaS platform.

5.1 A scheduling model of node-level FaaS execution

In this section, we define the optimization problem of minimizing latency-related objectives in the FaaS environment. The aim of this problem is to be simply defined, yet realistic enough to address dilemmas encountered in the serverless practice. Our notation follows the standard of Brucker [48] and common definitions from Section 2.1.

We consider a single physical machine with m parallel processors/cores $P_1, P_2, \ldots, P_m$ (a processor is a standard scheduling term; our processor maps to a single core on the machine). This machine has been assigned a set of n stateless functions, $f_1, f_2, \ldots, f_n$, that can be invoked multiple times, without a significant startup time (i.e., they are already loaded into memory). The functions are stateless, so one processor can execute one function at a time, but at any moment invocations of the same function can be independently processed on different processors. Each invocation (call) corresponds to a single end-user request. The actual execution time of f_j differs between calls, and we model it as a random variable with the $\mathbb{P}_j$ distribution. We consider both a preemptive and a non-preemptive case. In the preemptive case, a process executing a function can be suspended by the operating system, and later restored on the same or on another processor. In the non-preemptive case, a process executing a function—once started—occupies the processor until the function finishes and the result is ready to be returned to the end user.

We assume that the $f_1, f_2, \ldots, f_n$ functions, assigned to the machine, have been selected by the controller. In the case of cloud clusters, where thousands of physical machines work simultaneously, this process may take into account complex placement policies (balancing the overall load, affinity to reduce cluster network load, anti-affinity to increase reliability by placing instances executing the same function on different nodes or racks). In this chapter, we focus on the micro-scale of a single node in such a cluster.

We consider a time frame of $[0, T)$ where T is a positive integer. Each function f_j can be executed multiple times in response to numerous calls incoming in this time frame. Thus, the instance of the considered problem can be described as a sequence of an unknown number of invocations (calls) in time. Let the i-th call be represented by a pair of values: the moment of call $r(i)$ and a reference $f(i)$ to the invoked function. Of course, $0 \leq r(i-1) \leq r(i) < T$ for all $i > 1$.

Once the i-th invocation is finished (e.g., the result is returned to the end-user), we know the moment of its completion $c(i)$, and the total processing time $p(i)$ that the invocation required. Note that it is possible that $c(i) \geq T$. We use two base metrics to measure the performance of handling a call. The flow time $F(i)$, i.e. $c(i) - r(i)$, corresponds to the server-side processing delay of the query. The stretch (also called the slowdown) $S(i)$, i.e. $F(i)/p(i)$, weights the flow time by the processing time.

For any instance of our problem, processes executing functions need to be continuously assigned to processors in response to incoming calls. We determine the quality of the obtained schedule based on the following performance metrics that aggregate flow times or stretches across all the calls.

- **Average flow time** (AF), $\sum_i F(i)/\#\{i\}$, where $\#\{i\}$ is a total number of invocations. It is a standard performance metric considered for over four decades in various industrial applications [121]. It corresponds to the average response time.
- **Average stretch** (AS), $\sum_i S(i)/\#\{i\}$, which takes into account the actual execution time of a call [117], and thus responds to the observation that it is less noticeable that a 2-second call is delayed by 40 milliseconds (with the stretch of $2.04/2 = 1.02$) than it would be in case of a 10-millisecond call (resulting in the stretch of $50/10 = 5$).
- **99th percentile of flow time** (F99), $x\colon \mathbb{P}(F(i) < x) = 0.99,$ and
- **99th percentile of stretch** (S99), $x\colon \mathbb{P}\left(S(i) < x\right) = 0.99$, which are less fragile variants of the maximum performance metrics [117]. As in this chapter we analyse on impact of different scheduling methods on end-user experience, we state that these metrics are more appropriate than maximum-defined metrics, as if the flow time or stretch of a call exceeds a perceptual threshold accepted by the end-user, the call is canceled and the function is called again (e.g., by refreshing a webpage). This perceptual threshold reduces the number of calls with an unacceptably high flow time that have a significant impact on the overall performance [122]. Our robust variants, measuring the 99th percentile, return a value x such that 99% of all invocations have stretch (or flow time) smaller than x.
- **Average function-aggregated flow time** (FF),

  $$\frac{1}{n}\sum_{j=1}^{n}\frac{\sum_{\{i\colon f(i)=j\}} F(i)}{\#\{i\colon f(i)=j\}},$$

  where $\#\{i\colon f(i) = j\}$ is the number of all f_j calls, and

- **Average function-aggregated stretch** (FS),

$$\frac{1}{n}\sum_{j=1}^{n}\frac{\sum_{\{i:\, f(i)=j\}} F(i)}{\sum_{\{i:\, f(i)=j\}} p(i)},$$

  which we propose as new metrics specific for the FaaS environment. Our aim is to measure the fairness of a schedule based on the average values of the flow time or stretch within the sets of invocations of the same functions. These metrics take into account that functions developed by different users may require different amounts of resources (i.e., time) and that the performance of an invocation should not depend significantly on the set of functions that share the same machine.

As the actual processing times of invocations are random variables, we solve a set of online stochastic scheduling problems. Using the extended three-field notation [48], we denote these problems as: $\mathrm{P}m|\text{on-line}, r(i), p(i) \sim \mathbb{P}_{f(i)}|\mathbb{E}[\sigma]$ and $\mathrm{P}m|\text{on-line}, \text{pmtn}, r(i), p(i) \sim \mathbb{P}_{f(i)}|\mathbb{E}[\sigma]$ where $\sigma \in \{\mathrm{AF}, \mathrm{AS}, \mathrm{F99}, \mathrm{S99}, \mathrm{FF}, \mathrm{FS}\}$.

Proposition 1. The $\mathrm{P}m|\text{pmtn}, r(i)|\mathrm{FF}$ and $1|r(i)|\mathrm{FF}$ problems are strongly $\mathcal{NP}$-Hard. The $1|r(i)|\mathrm{FS}$ problem is $\mathcal{NP}$-Hard.

Proof. Consider special cases of the above problems in which each function is invoked exactly once. Then, the FF metric becomes equivalent to the $\sum_i c(i)$ metric. The $\mathrm{P}m|\text{pmtn}, r(i)|\sum_i c(i)$ [123] and $1|r(i)|\sum_i c(i)$ [124] problems are strongly $\mathcal{NP}$-Hard. Similarly, the FS metric becomes equivalent to the $\sum_i S(i)$ metric. It is known that the $1|r(i)|\sum_i S(i)$ problem is $\mathcal{NP}$-Hard [125]. □

5.2 Measuring invocations in the Azure dataset

The recently published Azure Function Trace [75] provides information about function invocations collected over a continuous 14-day period between July 15th and July 28th, 2019. For each day within this period, the trace presents a number of invocations of each of the monitored functions during each minute of the day ($24 \cdot 60 = 1440$ separate measurements). We denote the number of invocations of the f_j function within the k-th minute of the trace as λ_j^k. The trace distinguishes between different invocation sources, e.g. incoming HTTP requests or periodic executions (`cron` tasks). In this chapter, we consider HTTP requests only, as they are less predictable. Additionally,

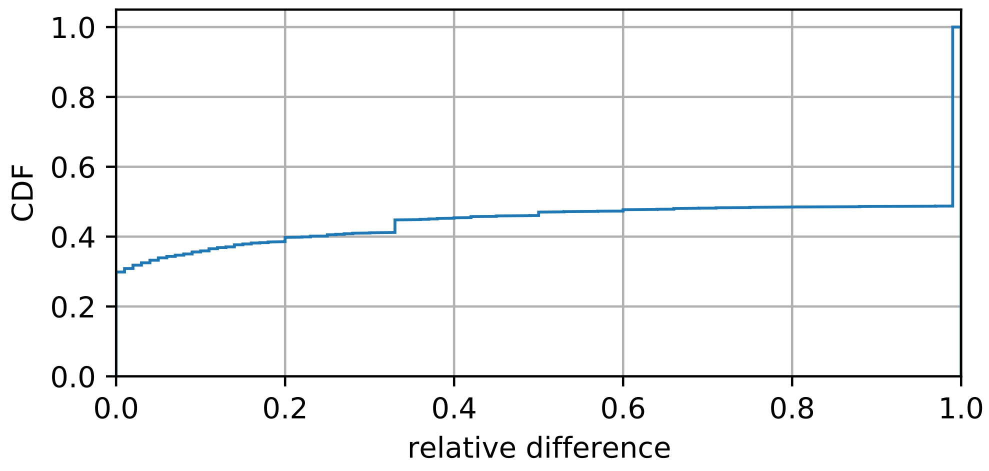


Figure 5.1: CDF of the relative difference Δ_j^k of the number of invocations compared to the previous one-minute interval over 10 000 randomly sampled interval-function pairs (k, j), where at least one of the λ_j^k and λ_j^{k-1} values is non-zero.

the trace shows the distribution of the execution times for each function during each day (based on weighted averages from 30-second intervals). For each function, the trace provides values of the 0th, 1st, 25th, 50th, 75th, 99th and the 100th percentile of this approximate distribution.

Some of the scheduling algorithms presented in Section 5.4 use the expected number of invocations in the k-th interval, λ_j^k, to make decisions online. The most straightforward way to estimate the unknown λ_j^k is to use the number of invocations in the previous interval, λ_j^{k-1}. We analyzed how the actual number of invocations differed between two consecutive one-minute intervals. In particular, we define the *relative difference* between λ_j^k and λ_j^{k-1} as

$$\Delta_j^k = \begin{cases} 0 & \text{if } \lambda_j^k = \lambda_j^{k-1} = 0 \text{ or } \lambda_j^{k-1} \text{ not known} \\ \frac{|\lambda_j^k - \lambda_j^{k-1}|}{\lambda_j^k + \lambda_j^{k-1}} & \text{otherwise} \end{cases}$$

We calculated relative differences Δ_j^k for all the recorded functions f_j and minutes k. We found out that 85% of these were equal to zero (meaning that the number of invocations did not change): for 93% of these cases (and 79% of all) there were no invocations in both the minutes ($\lambda_j^k = \lambda_j^{k-1} = 0$). Next, we studied in detail the cases for which $\lambda_j^k + \lambda_j^{k-1} > 0$. We randomly sampled

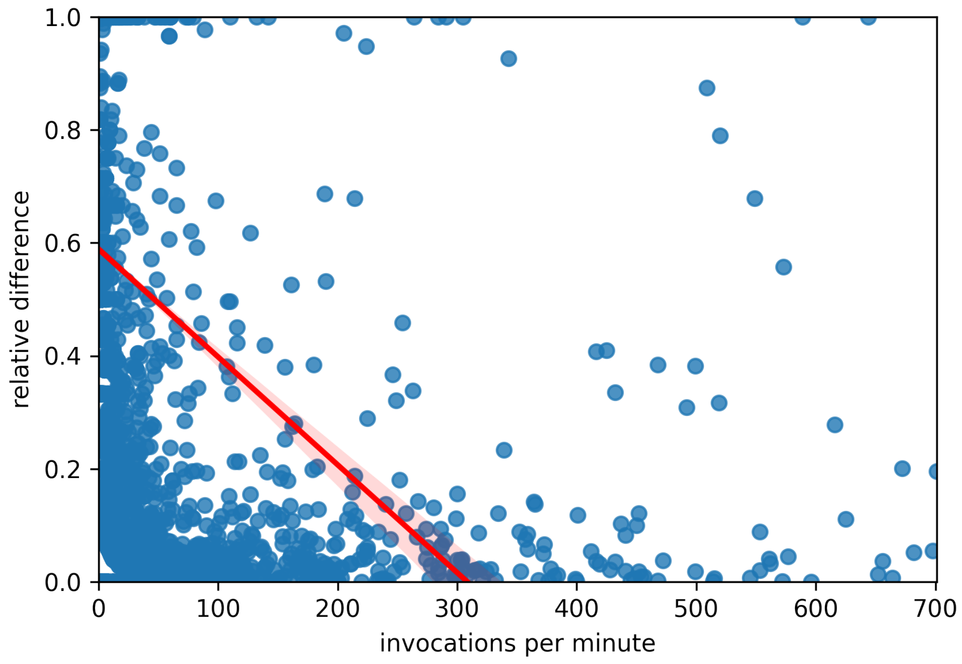


Figure 5.2: Relative differences for numbers of invocations compared to the previous one-minute interval. Each point shows one of 10 000 randomly sampled interval-function pairs (k, j), where at least one of the λ_j^k and λ_j^{k-1} values is non-zero. Red line represents linear regression fit to the visible data. Bands around the line indicate 90% confidence interval.

10 000 pairs (k, j) for which $\lambda_j^k + \lambda_j^{k-1} > 0$, and calculated the corresponding relative differences. Fig. 5.1 shows the CDF of the obtained Δ_j^k values. In 29% of the cases the number of invocations did not change; but for roughly 51% the relative difference was 1.0, denoting cases in which either λ_j^k or λ_j^{k-1} was 0. Fig. 5.2 shows Δ_j^k as a function of λ_j^k (for clarity of the presentation, the x-axis is cut at the 99th percentile of all λ_j^k values). We see that the relative difference decreases with the increased number of invocations per minute.

In further sections we present algorithms taking advantage of this knowledge to predict number of invocations to come. However, our algorithms additionally rely on the $p(i)$ values, the execution times of invocations. The analysis of $p(i)$ in the trace is presented in [75] (Section 3.4), thus we do not repeat it in this chapter.

5.3 Mapping the Azure dataset to our model

The theoretical model introduced in Section 5.1 is an on-line scheduling problem with release times and stochastic processing times. In this section, we show that monitoring data from Azure, a real-world FaaS system, is sufficient to fulfill the assumptions we take in the model (e.g., that the processing times are generated by an arbitrary distribution). The main issue is that when the number of events (e.g. function invocations) is large, it can be monitored only in aggregation—as in the case of the number of invocations and function execution times in the Azure dataset. Thus, we need to extrapolate these aggregations. Below, we describe how we acquire two sets of parameters required by our theoretical model: invocation times and the random distribution of processing times.

5.3.1 Invocation times

The Azure dataset does not give us the exact moments of invocation of each function. However, the total *number* λ_j^k of invocations of each function f_j is known for every k-th monitored minute. Thus, we assume that T, the duration of the considered time frame, is a multiple of 60 000 (number of milliseconds in a minute; our base unit is a millisecond because the processing times are given in milliseconds). The number of invocations, λ_j^k, may change in the $[0, T)$ interval as, for example, some functions are called intensively in the morning and rarely during the night. In order to model such changes, we divide the $[0, T)$ time frame into $K = T/60\,000$ consecutive, one-minute intervals v^k:

$$\begin{aligned}
&v^1 = [0, 60\,000),\\
&v^2 = [60\,000, 120\,000),\\
&\dots\\
&v^K = [(K-1)\cdot 60\,000, T).
\end{aligned}$$

We use the values of λ_j^k obtained directly from the Azure dataset to generate invocation times of function f_j in these intervals. Following a standard queueing theory, we assume that in the interval v^k, each function f_j is called based on the Poisson point process with rate $\lambda_j^k/60\,000$. Thus, the time (in milliseconds) between consecutive calls of function f_j in the v^k interval is a random variable with the $\mathrm{Exp}(\lambda_j^k/60\,000)$ distribution. Our scheduling model does not rely on this assumption—we use $r(i)$ values, the realizations of the random variables. However, some of our scheduling algorithms estimate λ_j^k for better scheduling decisions.

5.3.2 Processing times

Precise information on $p(i)$ values, the execution times of single invocations, is not provided in the Azure dataset. Our model assumes the existence of the distribution $\mathbb{P}_j$ (of execution times of a function f_j) in its exact form. However, the Azure dataset only shows selected percentiles of the empirical cumulative distribution function (p_0, percentile 0, p_1, the 1st percentile, p_{25}, p_{50}, p_{75}, and p_{100}). We thus approximate $\mathbb{F}_j$, the CDF of $\mathbb{P}_j$, by a piecewise-linear interpolation of these percentiles. For example, if $x \in (p_1, p_{25}]$, then

$$\mathbb{F}_j(x) = 0.01 + 0.24 \cdot \frac{x - p_1}{(p_{25} - p_1)}.$$

The actual processing time of each invocation is generated from $\mathbb{F}_j$.

5.4 Scheduling algorithms

Within the mapping described in the previous section, we apply both the well-known and theoretically-grounded strategies, and new approaches. In particular, we consider the following strategies.

- **FIFO** (*First In, First Out*, for a non-preemptive case) — all invocations are queued in the order in which they were received. When a processor is available, it is assigned the invocation with the lowest value of $r(i)$.
- **SEPT** (*Shortest Expected Processing Time*, for a non-preemptive case) — when a processor is available, it is assigned the invocation with the shortest expected processing time.
- **FC#** (*Fair Choice* based on the number of invocations, for a preemptive and a non-preemptive case) — when a processor is available, it is assigned the invocation which is the most unexpected. In fact, we want functions that are called occasionally to have larger priority than the frequently-invoked ones. At any point $t \in v^k$, the most unexpected invocation is the one with the lowest value of $\max\{\lambda_j^k, \#\{i \colon r(i) \in v^k \text{ and } f(i) = j\}\}$. The priority is thus determined based on the maximum of the expected number of invocations in the considered period and the actual number of these invocations.
- **FCP** (*Fair Choice* based on the total processing time, for a preemptive and a non-preemptive case) — when a processor is available, it is assigned the invocation related to the least demanding function in total. In fact, we want functions that use limited resources to have

larger priority than the burdening ones. At any point $t \in v^k$, the least demanding invocation is the one with the lowest value of

$$\max \left\{ \lambda_j^k \cdot \mathbb{E}[X \sim \mathbb{P}_j], \sum_{\{i\colon r(i) \in v^k \text{ and } f(i) = f_j\}} p(i) \right\}.$$

The priority is thus determined based on the maximum of the expected total processing times of invocations in the considered one-minute interval and the actual value of this amount.

- **RR** (*Round-robin*, for a preemptive case) — all invocations that have not been completed are queued. When a processor is available, it is assigned the first invocation from the queue. If the execution does not complete by a fixed period of time is it preempted and moved to the end of the queue. We consider periods of the length of 10, 100 and 1000 milliseconds. As this strategy is used as a reference, we assume that all the invocations have the same priority.

- **SERPT** (*Shortest Expected Remaining Processing Time*, for a preemptive case) — when a processor is available, it is assigned the invocation with the shortest expected remaining processing time. The strategy is applied only when an invocation is finished or a new call is received, even if all the processors are busy. Such a restriction is introduced because otherwise executions of functions with increasing expected remaining processing times could be preempted an arbitrarily large number of times.

In all cases, if the rule is unambiguous, we select the invocation with the lowest value of $r(i)$. The expected (remaining) processing times and the values of λ_j^k are estimated based on previous invocations. For each of SEPT, SERPT, FC# and FCP strategies, we introduce two methods: Position and Update. The first method returns the position of an invocation in the execution queue. For example, in case of SEPT, it returns the expected processing time of a function. The Update method is called after the execution of the invocation ends, and it updates auxiliary data structures that are used by the Position method. The framework algorithm (Alg. 4) for the above strategies is based on a standard scheduling loop. In this loop, we prioritize calls and choose the one with the lowest position.

The positions of invocations are calculated differently for different strategies. We present pseudocodes for two of them, a non-preemptive SEPT and a preemptive SERPT, as FC# and FCP are similar. Alg. 5 defines the Position and Update methods for the case of the SEPT strategy. As the invoked

Algorithm 4 A framework scheduling algorithm

```
 1: $Q \leftarrow \{\}$                                    ▷ A queue of incoming invocations
 2: $A \leftarrow \{\}$                                    ▷ A set of acknowledged invocations
 3: $E \leftarrow \{\}$                                    ▷ A set of invocations being processed
 4: preemptive $\in \{\text{true}, \text{false}\}$          ▷ Is the strategy preemptive?
 5: **while** true **do**
 6:     **wait until** (a processor is free **and** $(|A| + |Q| > 0)$) \
            **or** (preemptive **and** $|Q| > 0$)
 7:     **for each** $i$-th call in $E$ that has been finished **do**
 8:         Update$(f(i), p(i))$
 9:         Remove $i$ from $E$
10:     Move all the invocations from $Q$ to $A$
11:     **if** preemptive **then**
12:         Move all the invocations from $E$ to $A$
13:     **while** $|A| > 0$ **and** there are free processors **do**
14:         $i' \leftarrow \arg\min_{i \in A}$ Position$(f(i), p(i))$
15:                                          ▷ Here, $p(i)$ is a partial processing time
16:         Assign the $i'$-th call to any free processor
17:         Move $i'$ from $A$ to $E$
```

function is known at the moment of the call, the methods are similar for all the functions, but the auxiliary data structures are function-dependent. At any point in time, we store two values for each function f_j: the total processing time (TPT_j) of all its previous invocations and the number of these invocations (NOC_j). The expected processing time is approximated using a standard estimator, the average execution time $\text{TPT}_j/\text{NOC}_j$.

Similarly, Alg. 6 provides the same methods for the SERPT strategy. For each function f_j, we store execution times of its previous invocations in the PT_j vector. In general, this vector can be arbitrarily long. However, to reduce the memory footprint of the algorithm, we might want to limit its size. In such a case, the oldest values can be replaced with the newest ones. This approach has two main advantages: (1) if the distribution of processing times changes in time, it can be reflected within the algorithm, (2) the memory is saved. On the other hand, if the number of remembered values is limited, the accuracy of the estimation of the expected remaining processing time (ERPT) is limited too. For clarity of the presentation, we omit most of the implementation details. For example, one can use binary search or priority queues to improve the complexity of the presented approach.

In order to determine the ERPT of a call of f_j after p milliseconds, we select from PT_j the execution times that were equal to or exceeded p. We

Algorithm 5 SEPT

```
1: for $j \in \{1, 2, \ldots, n\}$ do                          ▷ Set initial values
2:     $\mathrm{TPT}_j, \mathrm{NOC}_j \leftarrow 0, 0$
3: procedure Position($j$, $p$)                                 ▷ $p$ is always 0
4:     if $\mathrm{NOC}_j > 0$ then return $\mathrm{TPT}_j / \mathrm{NOC}_j$
5:     else if $\sum_j \mathrm{NOC}_j = 0$ then return 0
6:     else return $\sum_j \mathrm{TPT}_j / \sum_j \mathrm{NOC}_j$
7: procedure Update($j, p$)
8:     $\mathrm{TPT}_j \leftarrow \mathrm{TPT}_j + p$
9:     $\mathrm{NOC}_j \leftarrow \mathrm{NOC}_j + 1$
```

then estimate the expected processing time of the current invocation based on the standard estimator, similar to the one presented in the case of SEPT.

5.5 Evaluation

We evaluate and compare our algorithms using discrete-time simulations, as this method enables us to perform evaluations on a large scale. We implemented the simulator and all the algorithms in C++ and validated its functionality using unit tests and a close-up, manual inspection of results on small instances. In this section, we present the results of performed evaluations. In Section 5.5.1, we present estimation models used with the algorithms. To make sure that our input data matches real-world scenarios, we generate test instances based on the Azure Functions Trace, with mapping indicated in Section 5.3. In Section 5.5.2, we describe data preprocessing, and in Section 5.5.3 we describe how we create inputs for our simulator. Finally, in Section 5.5.4–5.5.5, we analyse the results of the simulations and the behavior of the tested algorithms.

5.5.1 Estimations and baselines

Our algorithms rely heavily on probabilistic estimations of parameters (e.g., the processing time $p(i)$); these in turn depend on estimations of parameters of the generating distributions (e.g., $\mathbb{E}[X \sim \mathbb{P}_j]$). The methods we use are simple. To measure how much we loose with this simplicity, we compare our methods with the ground truth on two different levels.

First, we compare our probabilistic methods with exact *clairvoyant* algorithms that rely on the knowledge of the true execution time (that the

Algorithm 6 SERPT

```
1: for $j \in \{1, 2, \ldots, n\}$ do                          ▷ Set initial values
2:     $\mathrm{PT}_j \leftarrow \{\}$                          ▷ A set of previous processing times
3: procedure POSITION($j$, $p$)
4:     $pp, pc \leftarrow 0, 0$
5:     for $pt \in \{x \in \mathrm{PT}_j : x - p \geq 0\}$ do
6:         $pp \leftarrow pp + (pt - p)$
7:         $pc \leftarrow pc + 1$
8:     if $pc > 0$ then return $pp/pc$
9:     else
10:        for $pt \in \{x \in \cup_j \mathrm{PT}_j : x - p \geq 0\}$ do
11:            $pp \leftarrow pp + (pt - p)$
12:            $pc \leftarrow pc + 1$
13:        if $pc > 0$ then return $pp/pc$
14:        else return 0
15: procedure UPDATE($j, p$)
16:     Add $p$ to $\mathrm{PT}_j$
```

real-world scheduler clearly does not have): the SPT (*Shortest Processing Time*) strategy in the non-preemptive case and the SRPT (*Shortest Remaining Processing Time*) strategy in the preemptive case. These two non-real strategies are based on full knowledge of the actual execution time of invocations that are not yet finished. This way we can analyze how strategies based on expectations approach these standard theoretically-grounded strategies for fixed processing times—and thus the limits of how much the algorithms can further gain from better estimates.

Second, our probabilistic methods estimate the parameters of distributions. For each invocation of function f_j, algorithms presented in Section 5.4 may require information about its expected (remaining) processing time or the expected number of invocations of function f_j in the current one-minute interval. As we want to measure the influence of the imperfection of such estimations, some of the algorithms are compared in three different variants: the *reactionary* one (RE), a *limited reactionary* one (RE-LIM), and the *foresight* one (FOR).

In the *reactionary* variant, the EPT, ERPT and λ_j^k values for the f_j function are not known and thus are estimated based on *all* the previous invocations of the function f_j. The λ_j^k values in the reactionary model are predicted a priori based on the actual number of invocations in the previous one-minute interval v^{k-1}, i.e.

$$\lambda_j^k = \begin{cases} 1 & \text{if } k = 1 \\ \#\{i\colon f(i) = j \text{ and } r(i) \in v^{k-1}\} & \text{if } k > 1 \end{cases}$$

The EPT and ERPT values are estimated as shown in Alg. 5–6. In particular, if it is not possible to estimate any of these values (e.g., there were no previous calls of a particular function), an arbitrary default value is used. For example, SEPT uses the average processing time of all previous invocations.

Although the storage needed to estimate the expected processing time of an invocation (see Alg. 5) does not depend on the number of invocations, this is not a case when the ERPT value is calculated (see Alg. 6). In a real system, keeping information about all the previous calls of any function f_j is not practical. Therefore, we introduce *limited reactionary variants* (RE-LIM) of some algorithms, in which we keep information about at most 1 000 last invocations of each function f_j. (We state that maintaining this additional information on modern computers has a negligible impact on computational capability. We also tested RE-LIM limited to 10 and to 100 executions which resulted in significantly worse performance.)

In order to check how better estimators would impact the performance of algorithms, we compare the algorithms against *foresight* (FOR) variants which use actual parameters of the distributions used to generate the instance. These parameters correspond to the perfect, clairvoyant estimations. More formally, in the foresight variants we assume that for each function f_j the values of $\mathbb{E}[X \sim \mathbb{P}_j]$ (expected processing times), $\mathbb{E}[X \sim \mathbb{P}_j | X \geq p] - p$ (expected remaining processing times) and λ_j^k are estimated a priori based on the whole instance. However, the algorithms are still probabilistic, e.g., for a just-released job we know its expected processing time, $\mathbb{E}[X \sim \mathbb{P}_j]$, but not the actual processing time $p(j)$.

5.5.2 Preprocessing of the trace data

We pre-processed the Azure dataset as follows. First, we filtered out 38 functions having multiple records per day, leaving 671 404 out of 671 080 records from all the 14 days. Then, as indicated in Section 5.2, we removed all the records that were not related to HTTP invocations, which further narrowed the dataset to 200 194 records. We were particularly interested in functions invoked by HTTP requests, as they are less regular and their invocation patterns are harder to predict and optimize — contrary to functions triggered by internal events (e.g. `cron` tasks). Finally, we omitted functions containing missing data (i.e., missing information about execution times), which resulted in 199 524 records of data on 30 325 individual functions.

Algorithm 7 Instance generation (based on Azure)

```
1:  function FILL($T_1$, $T_2$, $m$, $\chi$, $\varepsilon$)
2:      $F \leftarrow \{1, 2, \ldots, n\}$
3:      $I \leftarrow \{\}$                                        ▷ Generated instance
4:      $L \leftarrow 0$                                           ▷ Total load
5:      do
6:          $j \leftarrow$ RANDOM($F$)
7:          $I_j \leftarrow$ seq. of invocation of the $f_j$ function
8:          $L_j \leftarrow$ total load of $I_j$
9:          if $L + L_j \leq (1 + \varepsilon) \cdot \chi \cdot m \cdot (T_2 - T_1)$ then
10:             $S \leftarrow S \cup \{j\}$
11:             $L \leftarrow L + L_j$
12:             $I \leftarrow I \cup I_j$
13:         $F \leftarrow F \setminus \{j\}$
14:     while $L < \chi \cdot m \cdot (T_2 - T_1) \wedge |F| > 0$
15:     return $I$
```

In general, functions in FaaS system may perform I/O operations. As the trace did not provide any information on either the I/O-intensiveness of individual functions, or the characteristics of I/O devices used in clusters, we assumed that functions are CPU-intensive. The given function processing times include the time needed to perform all the I/O operations. Thus, in our simulations, a processor remains busy during I/O phases. However, on the kernel level, the job performing the I/O would change its state to "waiting" ("not ready") and another ready job would be assigned to the processor. As a consequence, our simulation results provide upper-bounds for what we could expect in the case of I/O-intensive functions.

5.5.3 Generating instances

The performance of the scheduling algorithms was tested for various configurations. Each configuration specified the number of available processors (10, 20, 50 or 100), their desired average load (70%, 80%, 90% or 100%) and the time frame duration T (30, 60 or 100 min).

For each configuration we generated 20 independent instances (each box shows statistics over 20 instances). For each instance, we randomly selected a window $[T_1, T_2)$ of length T within one of 14 days of the trace. This way, each instance was generated based on the data coming from a consistent interval in the trace. From within the $[T_1, T_2)$ window, we randomly selected a subset of functions so the average load χ is achieved for the given number

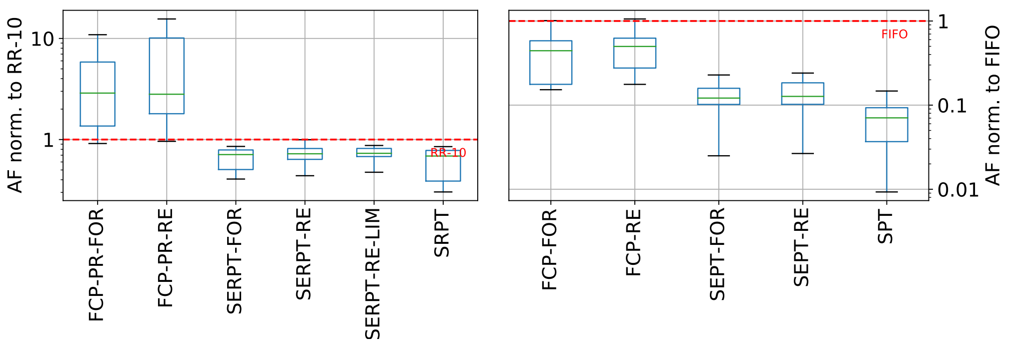
m of processors. Alg. 7 describes this process. First, we pick all functions having any invocation in the $[T_1, T_2)$ window (for clarity, we denote these functions by $1, 2, \dots, n$). Then, we randomly select functions from this set until the load of the generated instance is in the $[\chi, (1+\varepsilon)\chi]$ range (with $\varepsilon = 2\%$) or the set of available functions becomes empty. For a selected function f_j, we generate a sequence of its invocations using the provided information about the number of calls within each minute of the $[T_1, T_2)$ time frame and the percentiles of average execution times of its invocations during the day. This process is fully consistent with the mapping described in Section 5.3. The invocations of the f_j function are included into the generated instance if the total load after such inclusion does not exceed $(1+\varepsilon)\chi$ of the total available CPU time. From this point on, we map $[T_1, T_2)$ to $[0, T)$. The generated instance contains all the information required to evaluate the proposed algorithms – for the i-th invocation we provide: the moment of call $r(i)$, a reference $f(i)$ to the invoked function and the true processing time $p(i)$. We stress that the $p(i)$ value is not revealed to the reactionary variants of the proposed algorithms, so they are required to estimate them online.

5.5.4 Comparison of different algorithms

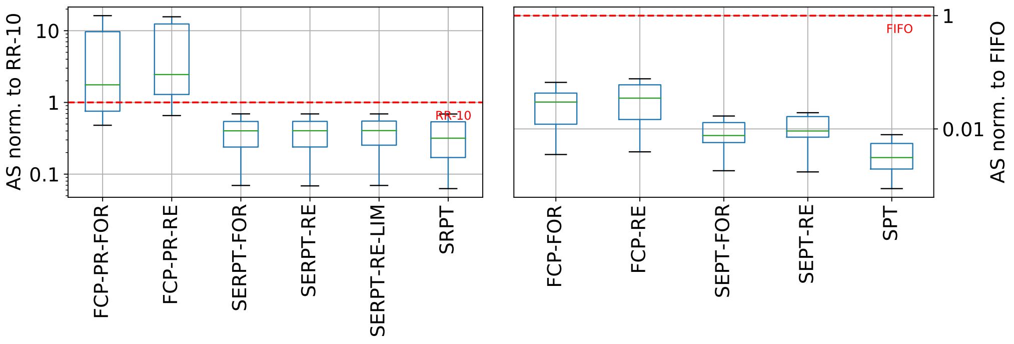
Fig. 5.3 presents the comparison of different metrics (formally defined in Section 5.1) for configuration of 20 processors, 90% average load and a 30-minute time frame T.

We split results into two groups: preemptive algorithms (left side of figures) and non-preemptive algorithms (right side). To mitigate the impact of the variability of results between instances, for each instance we *normalize* the performance metric (e.g., the average flow time) by the performance of a baseline algorithm. Results for preemptive algorithms are normalized to round-robin with a 10-millisecond period (denoted by RR-10), i.e., metric values for each test case are divided by corresponding results for RR-10. (We also tested round-robin variants with periods of 100 and 1 000 milliseconds, but they had worse results than RR-10 for all tested metrics, thus we skip them). Similarly, results for non-preemptive algorithms are normalized to FIFO. As RR-10 and FIFO always have normalized performance equal to 1, they are not shown on graphs.

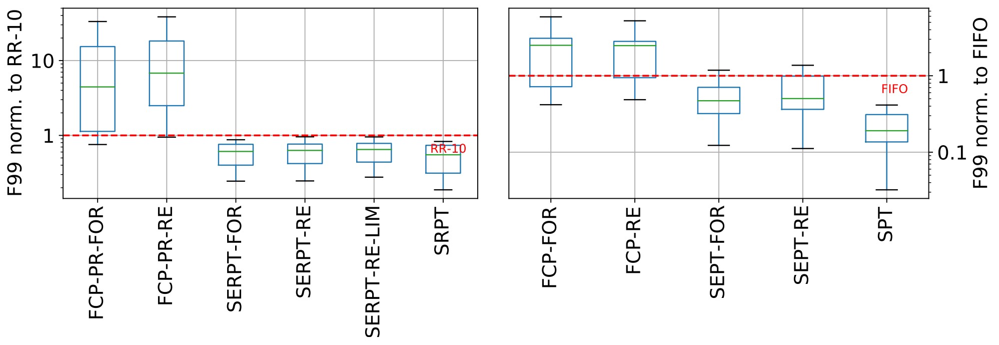
For all considered performance metrics, our proposed SERPT and SEPT algorithms significantly improve the results compared to the baselines, round-robin and FIFO. The smallest improvements are in flow-time related metrics for preemptive case (Fig. 5.3, (a) and (c), left)—but, as SERPT is close to the clairvoyant SRPT, we see that there is not much space for improvement. In the non-preemptive variants (Fig. 5.3, (a) and (c), right), the improvements



(a) Average flow time

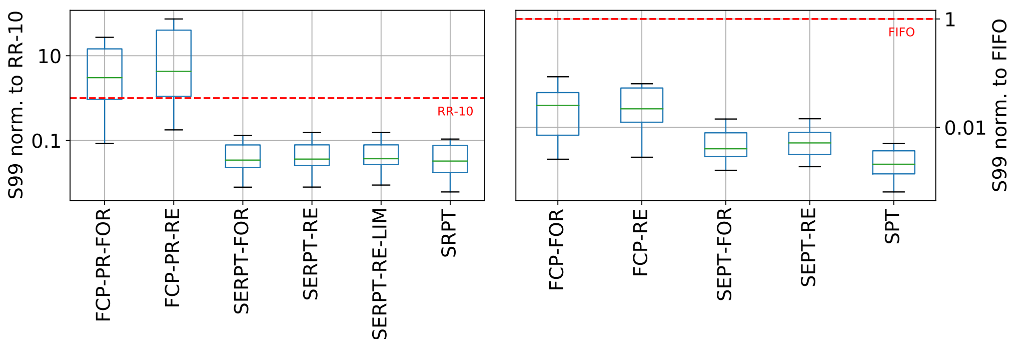


(b) Average stretch

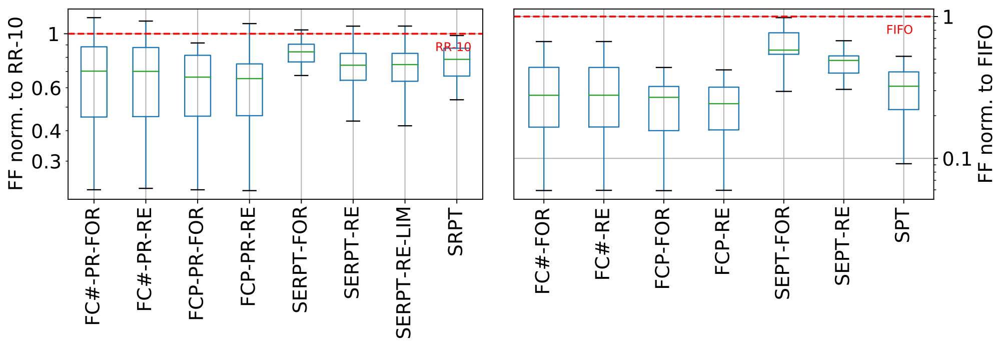


(c) 99th percentile of flow time

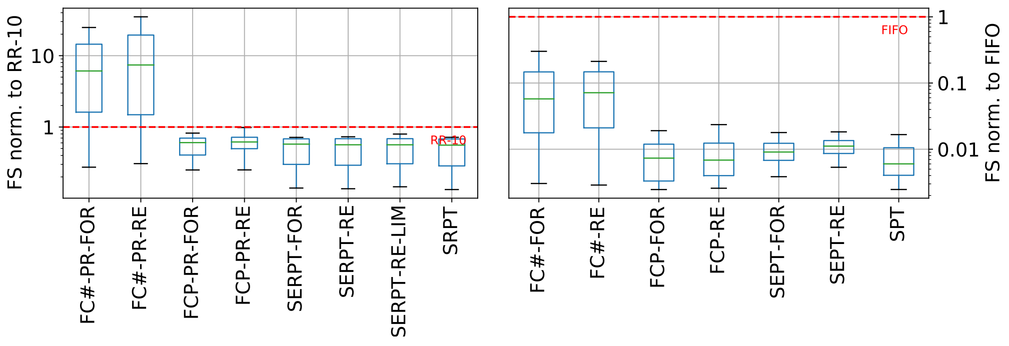
Figure 5.3: Comparison of different metrics. Each box shows statistics over 20 independent instances. Each instance has 20 processors, 30-minute time frame and 90% load. The red line indicates the baseline algorithm. (Continuation on the next page.)



(d) 99th percentile of stretch



(e) Average function-aggregated flow time



(f) Average function-aggregated stretch

Figure 5.3: (Continued.) Comparison of different metrics. Each box shows statistics over 20 independent instances. Each instance has 20 processors, 30-minute time frame and 90% load. The red line indicates the baseline algorithm.

in the average flow time are almost an order of magnitude. The reduction in stretch (Fig. 5.3, (b) and (d)) is larger: in non-preemptive variants by two orders of magnitude; in preemptive variants from 2-times for the average and to more than an order of magnitude for the 99th percentile. For all these metrics, SERPT in reactionary and foresight variants are close to SRPT, even though SRPT is clairvoyant while SERPT relies on estimates. This proves that our simple estimates of processing times are sufficient. However, in non-preemptive cases, the difference between SEPT and the clairvoyant SPT is larger: here, the impact of a wrong processing time estimate is harder to correct. SERPT limited to 1000 last executions (SERPT-RE-LIM) performed similarly to SERPT-RE for all tested metrics, which is promising, as that variant requires less memory when implemented in a real-world scheduler. For our fair, function-aggregated metrics (Fig. 5.3, (e) and (f)), FCP dominates other variants including FC# (which we skip from other figures as it was always dominated by FCP)— with the exception of average stretch in the clairvoyant variant, where FCP performance is similar to SERPT.

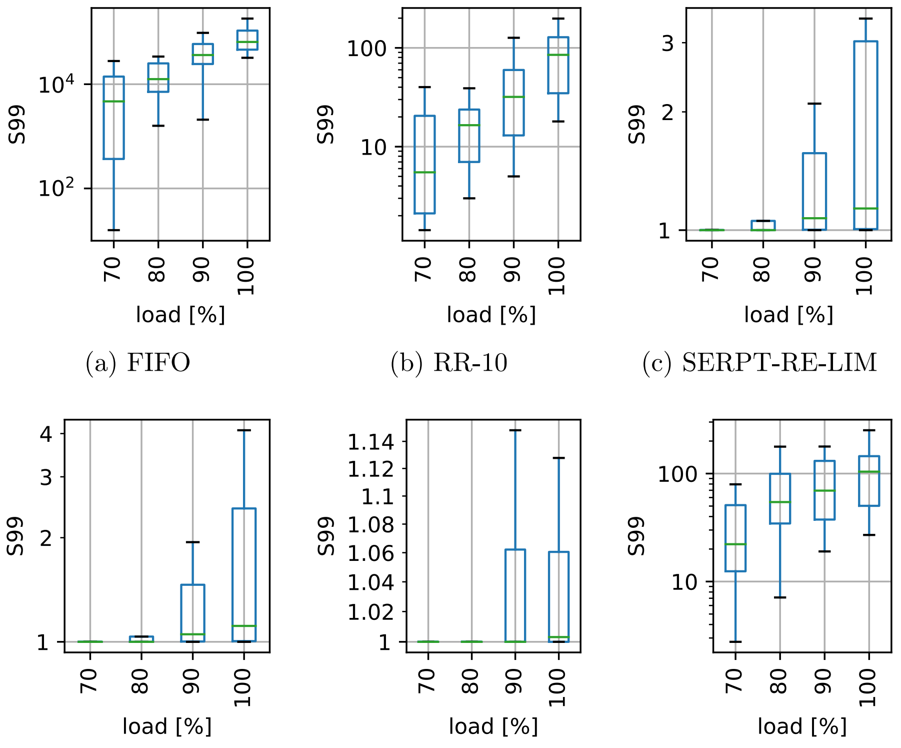
5.5.5 Impact of instance parameters

To make sure that the obtained results are valid for a wide range of scenarios, we verified the impact of changing average loads, processor counts and time window lengths.

Fig. 5.4 presents the 99th percentile of stretch (S99) with different loads. We chose this metric as it is the most sensitive to the density of function calls. First, for all loads SRPT results are close to the optimal 1, demonstrating that in all cases it is feasible to pack invocations almost optimally. Second, stretch increases with load for all other algorithms—however, both the increase and the absolute numbers are larger for the baselines FIFO and RR, compared with SERPT. Fig. 5.5 reinforces this observation: the higher the load, the better is the performance of our algorithms compared to the baselines.

We also analyzed how results change when the number of processors changes, with up to 100 processors (as the largest C2 instance in AWS has 96). Fig. 5.6 shows that with the increase in the number of processors, it is easier to schedule invocations almost optimally even with simple heuristics, as it is less and less probable that all processors will be blocked on processing long invocations—thus, the impact of better scheduling methods diminishes. This is confirmed by smaller relative gains of the SRPT in the preemptive case (Fig. 5.6, (f)).

Finally, we verified whether 30-minute time frame is reasonable by providing results that can be extrapolated on larger time windows. In Fig. 5.7



(a) FIFO (b) RR-10 (c) SERPT-RE-LIM

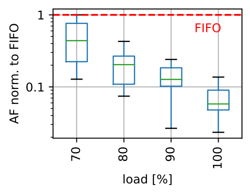
(d) SERPT-RE (e) SRPT (f) SPT

Figure 5.4: 99th percentile of stretch when varying the average load. 30-minute time frame, 20 processors.

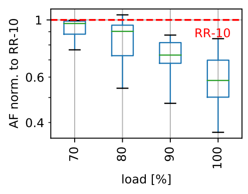
we present representative results on samples generated with different time frame durations. All the results are comparable, which indicates that our algorithms provide similar results for longer time spans.

5.6 Related Work

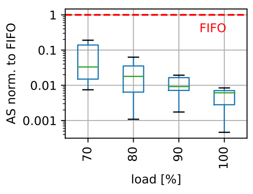
Scheduling with fixed processing times A number of theoretical research papers on scheduling with release dates was focused on fixed job execution times. As the release date of each job, r_i, is schedule-independent, the total flow-time objective, $\sum(C_i - r_i)$, becomes equivalent to the total completion time, $\sum C_i$. It was shown that although the $\mathrm{P}m||\sum C_i$ problem is polynomially-solvable [126], the $\mathrm{P}m|\mathrm{pmtn}, r_j|\sum C_i$ [123], [127] and even $1|r_i|\sum C_i$ [124] problems are strongly $\mathcal{NP}$-Hard. For these general problems, some online algorithms were analyzed. The modified version of the delayed SPT strategy [128]) was shown to provide a 2-competitive ra-



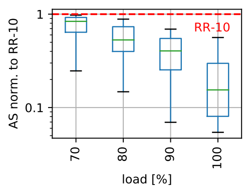
(a) SEPT-RE, average flow



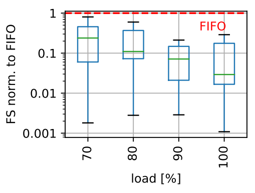
(b) SERPT-RE-LIM, average flow



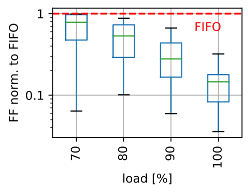
(c) SEPT-RE, average stretch



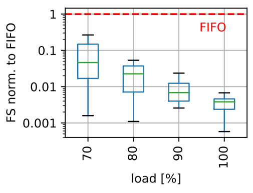
(d) SERPT-RE-LIM, average stretch



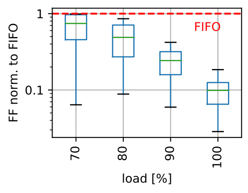
(e) FC#-RE, FS



(f) FC#-RE, FF

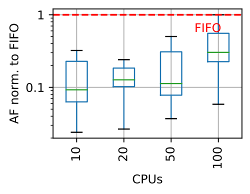


(g) FCP-RE, FS

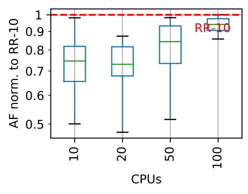


(h) FCP-RE, FF

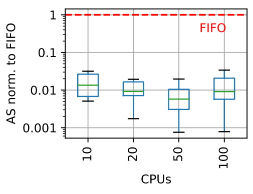
Figure 5.5: Relative performance when varying the average load. 30 min time frame, 20 processors.



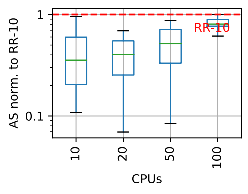
(a) SEPT-RE, average flow



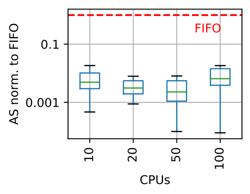
(b) SERPT-RE-LIM, average flow



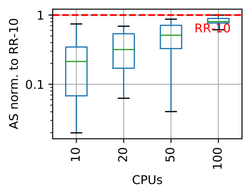
(c) SEPT-RE, average stretch



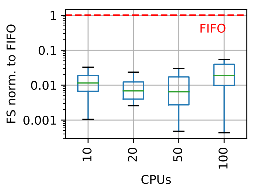
(d) SERPT-RE-LIM, average stretch



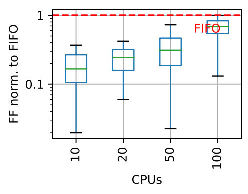
(e) SPT, average stretch



(f) SRPT, average stretch



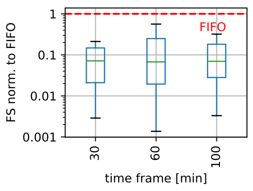
(g) FCP-RE + FS



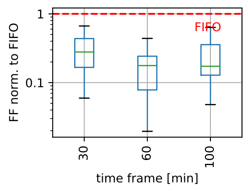
(h) FCP-RE + FF

Figure 5.6: Relative performance when varying the number of processors. 30-minute time frame and 90% load.

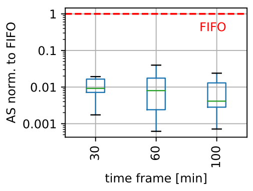
tio for the P$m|$on-line$, r_j|\sum C_i$ problem [129]. The same competitive ratio can be achieved for the P$m|$on-line, pmtn$, r_j|\sum w_i C_i$ problem [119], but only a 2.62-competitive algorithm is known for the non-preemptive case



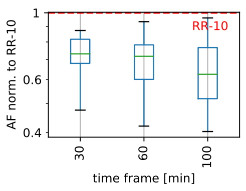
(a) FC#-RE + FS



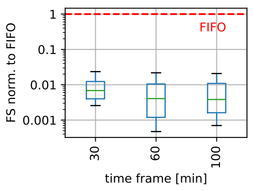
(b) FC#-RE + FF



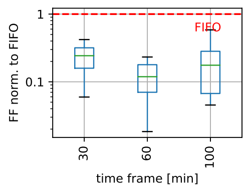
(c) SEPT-RE + AS



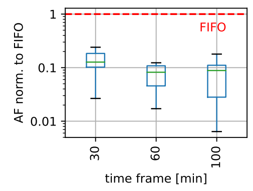
(d) SERPT-RE-LIM + AF



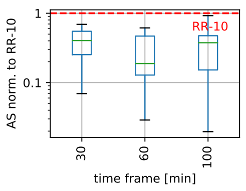
(e) FCP-RE + FS



(f) FCP-RE + FF



(g) SEPT-RE + AF



(h) SERPT-RE-LIM + AS

Figure 5.7: Comparison of the algorithms with different time frame durations. 20 processors and 90% load.

[130]. These results were extended in [131] where it was proven that, for $P = p_{\max}/p_{\min}$ being the ratio of the maximum to the minimum job execution time, SRPT is a $O(\log(\min\{\frac{n}{m}, P\}))$-approximation offline algorithm for the Pm|pmtn, $r_j|\sum C_i$ problem.

In the case of the total stretch ($\sum S_i$) objective, the results are even less promising. It was shown that the SRPT strategy is 14-competitive for the Pm|on-line, pmtn, $r_j|\sum S_i$ problem [118]. In [132], it was shown that there exists a strategy with a constant-factor competitive ratio for a uniprocessor machine even if the processing times are known only to be within a constant factor of accuracy. There was also shown a PTAS for the offline version of the 1|pmtn, $r_j|\sum S_i$ problem [132].

The maximum-defined objectives, such as $\max\{C_i - r_i\}$ ($F_{\max}$) and $\max\{(C_i - r_i)/p_i\}$ ($S_{\max}$), were also discussed. A number of results were shown in [117]. In particular, it was proven that for the Pm|on-line, $r_i|F_{\max}$ problem, FIFO is a $(3 - 2/m)$-competitive strategy, and this bound is tight. It is also known that the offline version of the $1|r_i|S_{\max}$ problem cannot be approximated in polynomial time to within a factor of $O(n^{1-\varepsilon})$, unless $\mathcal{P} = \mathcal{NP}$ [117]. Finally, it is shown that for $P = p_{\max}/p_{\min}$, every online algorithm for the Pm|on-line, pmtn, $r_i|S_{\max}$ problem is $\Omega(P^{1/3})$-competitive for three or more job sizes.

Stochastic approaches to scheduling The results for fixed job processing times provide us lower bounds for the expected performance in case of the corresponding stochastic problems. In practice, stochastic problems are more complex. For example, it was shown that the performance guarantee for the $\mathrm{P}m|\text{on-line}, r_j|\mathbb{E}[\sum w_i C_i]$ problem can be upper-bounded by $\frac{5+\sqrt{5}}{2} - 1/(2m)$, if the expected remaining processing time of any job is a function decreasing in time [133], compared to a 2.62-competitive ratio in case of a non-stochastic variant [130]. A good review of research papers considering stochastic scheduling problems, mostly non-preemptive, can be found in [134], [135]. It can be observed that most results are related to the $\sum w_i C_i$ objective.

To the best of our knowledge, there is only a limited number of papers on preemptive stochastic scheduling (e.g., [136], [137]). However, performance guarantees are shown only for specific distributions of processing times (i.e., discrete ones). Moreover, we found no theoretical papers on stochastic scheduling with average or maximum stretch as performance metrics.

5.7 Summary and discussion

The analyzed problem was driven by real-world data provided in the Azure Function Trace. We studied various non-clairvoyant, online scheduling strategies for a single node in a large FaaS cluster. Our aim was to improve performance measured with metrics related to response time or stretch of the function invocations. To estimate the values of the expected processing time or the expected remaining processing time of an invocation, we took advantage of the fact that the same function is usually invoked multiple times. This way, we were able to adapt SEPT and SERPT strategies with no significant increase in the consumption of memory or computational power. For our newly-introduced fair metrics, the function-aggregated stretch and flow time, we proposed two new heuristics, called Fair Choice. There, decisions are made based on an additional estimation of the expected number of function calls in the next monitoring interval.

Compared to round-robin and FIFO baselines, in the base case of our simulations, our proposed SEPT and SERPT strategies reduce the average flow time by a factor of 1.4 (preemptive) to 6 (non-preemptive); and the average stretch by a factor of 2.6 (preemptive) to 50 (non-preemptive). Gains over FIFO and round-robin increase with increased pressure of the workload on the system: with the lower number of processors and the higher average load. For the fair, function-aggregated metrics, our newly introduced Fair

Choice strategies clearly outperform other implementable algorithms when measuring the flow time (while the gain is smaller for stretch).

SEPT, SERPT and Fair Choice can be easily implemented in the node-level component of the FaaS scheduling stack (e.g., the Invoker module in OpenWhisk). We further explore these possibilities in the following chapter.

Chapter 6

System perspective on scheduling on a single node

In the approach introduced in Chapter 5, we reduce the response latency in a loaded FaaS system by optimizing the ordering of requests on a single worker node. We benefit from a feature of FaaS workloads: functions are short-lived (seconds) but called repeatedly. Thus, our scheduling strategies use estimates of the frequency and execution time of a call.

Our simulation results (Sections 5.5.4–5.5.5) show performance improvements and indicate which modifications are worth applying into the FaaS scheduler. In this chapter, we take one step closer to the function call scheduling in a FaaS system. First, we adapt selected algorithms from Chapter 5. Then, as from system's perspective it is crucial to ensure fairness between users, we propose two additional algorithms focusing on this issue, which still rely on information that can be gathered on a single node.

We implement our methods in the Apache OpenWhisk platform and evaluate them experimentally using FaaS workload derived from SeBS benchmark [54]. In contrast to unmodified OpenWhisk, in this approach, we do not rely on OS-level preemption in FaaS, as one of our goals is to avoid oversubscribing CPUs. Therefore, we set each executing container's CPU limit to a single core and we ensure that number of parallel calls does not exceed the number of available CPUs.

From the system's perspective, our approach has two main advantages. First, as our results show, with our node-level scheduling strategies, each worker node handles higher loads with significantly lower response time than the baseline method. Thus, our approach does not require so much CPU buffer to handle peak loads, so the amount of resources in the steady state can be reduced, resulting in lower infrastructure costs without compromising the latency. Second, we do not modify the other two resource managers of a typical FaaS installation: the controller (receiving the requests) and the load balancer (allocating individual invocations). Our method is thus orthogonal — and can be applied in addition to — the many recent optimization efforts that concentrated on these elements of FaaS infrastructure: function placement [18], load balancing [88], autoscaling [120], or choosing the appropriate repertoire of warm containers through predicting future calls and setting up containers in advance [75].

Although our scheduling strategies rely on estimates of the frequency and execution time of a call measured locally, we do not limit our analysis to a single node. By experiments, we show that these improvements naturally translate to a reduction of response-related metrics in a multi-node cluster.

This chapter is organized as follows:

- We present a scheduling model of OpenWhisk invoker (Section 6.1)
- We adapt method of node-level container management from Chapter 5 and introduce a number of scheduling policies that are based on locally-gathered historical data on function calls (Section 6.2).
- We implement our policies in Apache OpenWhisk. We conduct experiments on a FaaS benchmark SeBS [54], that we extend to handle OpenWhisk (Section 6.3).
- We analyze impact of cold starts and evictions on observed response times and system stability (Section 6.4). We show that this method reduces the number of preemptions (compared to interactive systems) and cold starts (compared to the baseline Apache OpenWhisk).
- We show that our policies improve response time metrics on the node (Section 6.5) and infrastructure (Section 6.6) level. In particular, compared to the baseline using 4 machines, our solution produces shorter response latencies running on just 3 machines.
- We summarize obtained results (Section 6.7).

As this work shows the systems' perspective on the model presented in Chapter 5, we present related work (from both theoretical and system view) in Section 5.6.

6.1 A scheduling model of an OpenWhisk invoker

Our node-level scheduling problem is formulated similarly to model in Section 5.1. In this chapter, we focus on heavy loads, i.e. we assume that in a small time window, the total number of function calls, $|I|$, exceeds the standard throughput of the node.

When user invokes a function call, the generated request has to pass through all system layers: the load balancer, the controller, internal message passing and the invoker. This takes non-negligible time – the $\hat{r}(i)$ denote the moment the request was generated by the *end-user*, $\hat{c}(i)$ indicate the moment when the results were obtained by the *end-client*. Previously, in Section 5.1 we defined $r(i)$ as the moment of arrival of the call at FaaS system, an $c(i)$ as the call's completion time.

The system is on-line and non-clairvoyant: a call is unknown until $r(i)$ when it arrives at FaaS system. Similarly, its processing time $\hat{p}(i)$ is known only when the execution ends and result is returned to the end-user.

Our goal is to minimize performance metrics related to response time in such an uncertain environment. A single call's *response time* (measured on client-side, contrary to flow time analyzed in Chapter 5) is $\hat{R}(i) = \hat{c}(i) - \hat{r}(i)$. This takes into account that both the request and the response are transferred through a potentially latent network (as opposed to measuring the flow time $F(i) = c(i) - r(i)$ just at the node level). We additionally measure the *client-side stretch* defined as $\hat{S}(i) = \hat{R}(i)/\hat{p}(i)$, i.e. response time expressed in units of the processing time.

To measure the system performance, we aggregate $\hat{R}(i)$ across all the calls. We will report the standard statistics: the *average client-side response time* $\sum_i \hat{R}(i)/|I|$, a metric widely used in operational research and systems [121]; the *average client-side stretch* $\sum_i \hat{S}(i)/|I|$; as well as order statistics: medians and quartiles.

Additionally, the processing time $\hat{p}(i)$ depends on (although it is not fully determined by) the function $f(i)$ being called, we will show aggregations of response time across all calls of the function $f(i)$. We do so to make sure that our methods do not discriminate against a certain class of function — short, long, often- or rarely-called.

6.2 Node-level scheduling policies

In Section 2.2 we described resource management in OpenWhisk. In our work we use the current default OpenWhisk policy as the baseline. However, the new action assignment (described at the end of Section 2.2) model, where controller is not responsible for selecting node to process request, is orthogonal to the node-level scheduling policies we present later in this chapter — our policies can be still used once this new model is fully implemented.

Our main goal is to improve the actual performance of a single FaaS node, and thus — by the rule of scaling — of a whole FaaS cluster. We replace the current approach (that uses free pool containers and FIFO queues) with more sophisticated — yet still greedy — scheduling policies, which are implemented using priority queues. The priority of an incoming action call may be determined by various environmental factors such as the actual processing times of similar actions performed in the past. We adapt approach presented in Chapter 5 and estimate the expected processing time of an action by the average processing time of at most 10 recent executions of the same action. Moreover, the processing time is estimated on the node-level and thus is not affected by network latency. Once a priority of a particular action call is computed, it does not change. Using the expected processing time as the action priority can starve some actions (if there is always a longer/shorter request waiting to be processed). It is not crucial for our strategies to not lead to starvation, as we consider only a short interval in which the system is overloaded. However, some of our strategies explicitly prevent starvation, because we determine the priorities based on the expected *completion* time or the total processing time from the past and calls are processed in increasing order of their priority value. Therefore, we introduce the three following strategies based on algorithms from Section 5.4:

- *First-In, First-Out* (FIFO), in which the action priority is the time $r(i) \geq \hat{r}(i)$ the action call is received by the invoker (pulled from a queue);
- *Shortest Expected Processing Time* (SEPT), in which the action priority is the expected processing time of the action, $\mathbb{E}(p(i))$. As in Section 5.4, we estimate $\mathbb{E}(p(i))$ as the average processing time $\bar{p}(j)$ over last 10 finished calls of the same function $f(i)$;
- *Fair-Choice* (FC), in which we prioritize actions based on the estimation of the processing time other of calls of the same function recently concluded. Namely, we define the priority of the i-th action as $\#(f(i), -T) \cdot \mathbb{E}(p(i))$, where $\#(f(i), -T)$ is the number of calls of function $f(i)$ during last T seconds, for T being a long time interval, e.g. 60 seconds. FC is related to FCP presented in Section 5.4.

We also introduce two new strategies preventing starvation:

- *Earliest Expected Completion Time* (EECT), in which the action priority is $r(i) + \mathbb{E}(p(i))$; this corresponds to the expected completion time of an action if a processor is immediately available;
- *Recent Expected Completion Time* (RECT), in which the priority is $r'(i) + \mathbb{E}(p(i))$, where $r'(i)$ is the moment when the previous call of function $f(i)$ was received;

For EECT, consider two actions, i-th and j-th. If $r(j) > r(i) + \mathbb{E}(p(i))$, then the j-th call will be executed after the i-th one. For this reason, it is impossible for the i-th call to wait infinitely long for being executed. The same reasoning can be applied to RECT, as the value of $r'(i)$ is increasing in time.

We stress that our scheduler dynamically estimates the expected processing time $\mathbb{E}(p(i))$ based on at most 10 most recent processing times of the same function $f(i)$ on the worker node. To present the results on stretch, we also use the run-time estimates based on off-line benchmarking of $f(i)$ (which we present in Table 6.1), but these off-line results are never used to make any scheduling decisions. This way, our approach remains valid for other functions, with processing times known a posteriori.

6.2.1 Towards non-preemptive function execution in OpenWhisk

OpenWhisk limits the number of busy containers (action containers that execute actions at any given moment) run by single node by the amount of available operational memory. One large container can be exchanged for a few smaller ones. By default, OpenWhisk assigns to each container a CPU share that is nearly linearly proportional to its memory requirement. Thus, it may occur that a container is assigned only a fraction of CPU time.

This policy leads to OS-level preemption. If the number of concurrently executed actions is greater than the number of CPU cores, then multiple context switches might be performed by the OS. Such context switching can have a significant negative impact on the response time.

As the theoretical variants of the strategies listed earlier are non-preemptive, we want to discourage the operating system from preempting currently executed actions. To achieve that, we drastically change the default approach:

- We limit the number of busy containers with the number of available CPU cores.
- We replace CPU limits based on memory requirements with fixed ones: a single container is always assigned a CPU limit of exactly one core.

As a direct consequence of such a change, we can state what follows. If the limit of busy containers is less or equal to the number of available CPU cores, we get close to a non-preemptive model. If the limit of busy containers is greater than the number of available CPU cores, then preemption is introduced by the operating system natively.

Using a limit on busy containers equal to the number of CPU cores seems more reasonable when one assumes that the executed actions are CPU-intensive. Otherwise, i.e. for I/O-intensive actions, some CPU cores may stay idle, although they could execute another function. As in the SeBS benchmark [54] we find both CPU- and I/O-intensive functions, we will verify the impact of that experimentally.

6.2.2 Implementation overview

We modified the source code of Apache OpenWhisk in order to implement the changes described above[1]. We stress that these modifications are orthogonal to the currently-developed changes in the action scheduling model [68].

Our policies need data on recent invocations of action calls. We gather this data by extending the invoker's processing pipeline. We log the moment a request is pulled from Kafka by the invoker to calculate the release time $r(i)$ (required by FC, EECT, and RECT). Then, when the invoker gets the response from the action container, we store the processing time in a per-function fixed-size buffer. We also replace the invoker's simple queue by a priority queue. The priorities are computed based on the scheduling policy (Section 6.2) selected at the start of each experiment (based on a new configuration option we added to OpenWhisk). The selected policy uses the data collected as above. For functions that were not executed yet, we estimate their execution time by zero.

To implement the CPU limits (Section 6.2.1), we changed the parameters of the *docker run* command used to create each action container. In our implementation, each container is always assigned a whole CPU core. Moreover, we modified the invoker's behavior, so there are no more concurrent calls than the number of available CPU cores.

6.3 Experimental setup

The goal of our experiments is to quantify the impact of the improved node-level scheduling policies. We thus use at least two separate OpenWhisk machines (physical in the on-premises experiments; virtual in the multi-machine experiments) — one for the controller, and at least one for the invoker with its action containers. Figure 6.1 shows a flow of a single request in our setup, for a single invoker.

[1]At the time of development, the most recent code available in OpenWhisk repository was commit `3802374d`

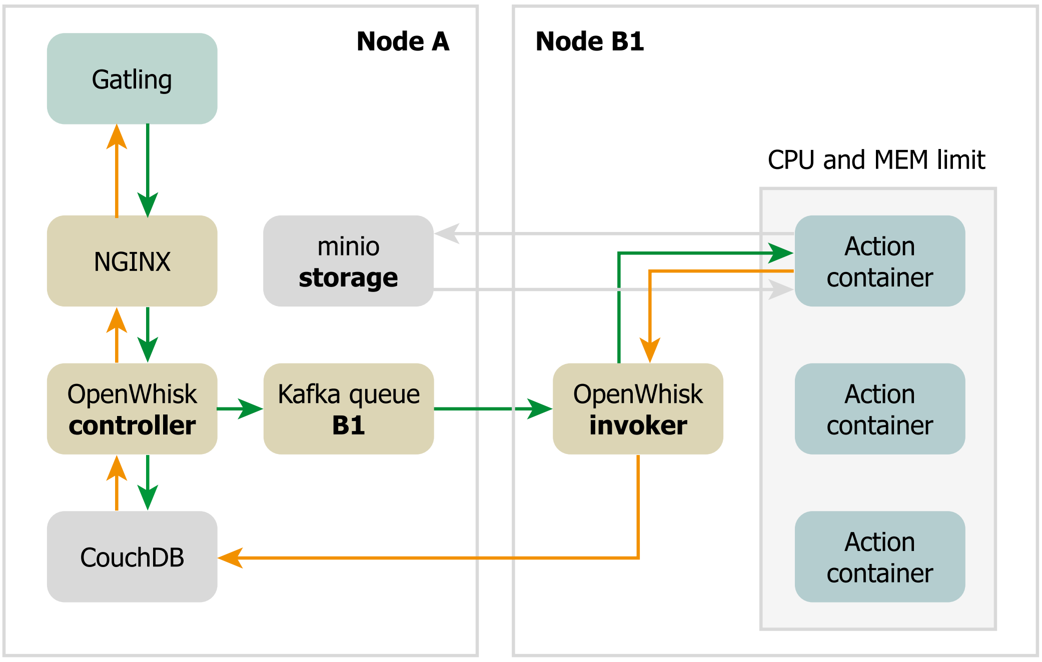


Figure 6.1: A flow of a single request generated by Gatling during our experiments.

We use functions from the SeBS [54] benchmark. Roughly half of these functions are computationally-intensive, while others strain I/O and network. SeBS is designed to measure commercial cloud services providers such as AWS Lambda or Microsoft Azure Serverless. We extend SeBS by designing and implementing wrappers allowing us to deploy SeBS on OpenWhisk.

We simulate end-clients by generating requests with the Gatling [138] load testing tool (to reduce network noise, Gatling is deployed on the same node as the OpenWhisk controller). We define test scenarios as sequences of function requests (see Section 6.3.1). Gatling executes such scenarios in a controlled and monitored manner. In contrast to other load testing tools, like Apache JMeter [139], Gatling allows us to use Scala to create, manage and execute multiple tests.

Our on-premises experiments use two machines: each machine has 256 GB of RAM and two Intel Xeon Silver 4210R CPUs @ 2.40GHz with 20 hyper-threaded cores. Our setup runs Ubuntu 20.04 (Linux kernel 5.11.0-34-generic) with Docker 20.10.7. We changed the default CPU frequency governors of both machines to *performance* to avoid CPU frequency scaling during experiments. We store Docker images and containers on NVMe. Our preliminary experiments used SSDs and the default 5.4.0-88-generic kernel, but we noticed Docker stability issues, especially for the OpenWhisk baseline, high load and 128 GB of OpenWhisk memory pool.

All FaaS requests are considered to be *blocking*, i.e. the request is sent over an HTTP connection which remains open until the result is returned to the client. If the system is overloaded, this time may increase significantly — by default OpenWhisk will return an error if it takes more than approx. 60s to execute the action. Thus, in an overloaded node with standard time limits, the "result" of a scheduling strategy is a tuple — response times and the number of failed requests. However, to be able to compare strategies directly, we strongly prefer to have a single measure. We thus decided to increase the time limits high enough to have zero time-outs. We claim this is fair as these increased limits have the same influence on all the measured strategies. Additionally, different FaaS providers set different timeouts on function duration (9 minutes in GCP and 15 minutes in AWS, while Azure Premium and Azure AppService allows arbitrary large timeouts). Moreover, not every function has to produce a response that is directly delivered to the end-user – in which case short timeouts are pointless.

6.3.1 The structure of an experiment

Our goal is to measure and compare the effectiveness of different scheduling policies fairly. The general structure of a single experiment is as follows.

First, we warm up the action containers by issuing a certain number of function calls. These calls initialize the action containers, thus reducing the influence of cold starts on the results (see Section 6.4 for a detailed discussion). If the number of available CPU cores is equal to c, then at most c containers for each function can be used simultaneously. This is so as the number of busy containers is limited by the number of available CPU cores (c.f. Section 6.2.1). Thus, we issue c parallel calls for each function. Note that we do not measure the response times of these warm-up calls. After the containers are warmed-up, we are ready to start the actual experiment.

We generate the measured load as a burst of requests that are all uniformly issued in a 60-second time window. These 60 seconds simulate a "difficult" overload scenario. With shorter (e.g. 10-second) bursts, scheduling inefficiencies will have a short-term influence on the response time and thus will not be that noticeable by the end-user. Conversely, longer bursts are best handled by a horizontal autoscaler adding more nodes. However, it takes at least dozens of seconds to set up a new, cold worker node — and then seconds to warm up the action containers.

After 60 seconds, no new requests are issued. Otherwise, e.g., if the intensity decreased gradually, it would be harder to measure the influence of the precise load intensity on the performance of the scheduling strategies.

After sending the last call to the OpenWhisk controller, Gatling waits until all the responses are returned. We report response times as measured by Gatling. We monitor the response errors to make sure that no (or very few) response errors are generated. In our initial experiments, such errors generally hinted to various system-level problems (e.g.: I/O bottlenecks on an SSD or Docker's inability to maintain many containers).

We use functions from SeBS [54], Table 6.1. We considered all the functions defined in the benchmark except the 3 Node.js implementations (we use their Python alternatives) and the network microbenchmarks (as they measure network latency). For each function, we defined a separate address of the HTTP endpoint used to invoke it, and specified its call parameters. To approximate function's processing time $p(i)$ for stretch-related metrics, we benchmarked each function in an idle on-premises setup: we warmed up the corresponding containers, and then we called this function 50 times. In our experiments, we do not want to measure the latency of the network, but the capacity of the FaaS node. On the other hand, we are unable to directly measure the duration of the execution of an action call on the invoker level. Thus, in our stretch metrics, instead of the processing time (denoted by $p(i)$), we use the median response time $\hat{p}(i)$ measured on the level of the Gatling client (see Table 6.1). Although it lets us compare the performance of the system in the context of different functions, such a change may result in stretch values that are less than 1.

6.3.2 Parameters: load intensity and CPU cores

Two configuration options significantly influence the results: (1) the available resources — in our case, the number of available CPU cores; (2) the load — in our case, the amount of requests issued during the 60-second window. Roughly, by doubling the amount of resources (e.g., increasing the number of cores from 10 to 20), the system should double the load it can handle. Thus, to meaningfully compare the impact of increased resources with constant load — or increased load with constant resources — we introduce the notion of *intensity* of a scenario: a multiplicative factor regulating the load (keeping the available resources constant).

Our experiments are designed so that each action is called the same number of times in a 60-second window. As we consider 11 functions, we want the total number of calls to be a multiple of 11. Thus, in a scenario of intensity v, we generate exactly $1.1 \cdot c \cdot v$ requests, where c is the number of CPU

Table 6.1: Functions from the SeBS benchmark tool measured on the client side, on-premises setup. The measurements include ca. 10 ms Kafka overhead.

	Response time		
Function name	5th perc.	Median	95th perc.
`dna-visualisation`	8 415 ms	**8 552 ms**	8 847 ms
`sleep` (1000 ms)	1 020 ms	**1 022 ms**	1 026 ms
`compression`	793 ms	**807 ms**	832 ms
`video-processing`	586 ms	**593 ms**	605 ms
`uploader`	184 ms	**192 ms**	405 ms
`image-recognition`	117 ms	**121 ms**	237 ms
`thumbnailer`	112 ms	**118 ms**	124 ms
`dynamic-html`	18 ms	**19 ms**	22 ms
`graph-pagerank`	11 ms	**12 ms**	15 ms
`graph-bfs`	11 ms	**12 ms**	13 ms
`graph-mst`	11 ms	**12 ms**	13 ms

cores for action containers. For example, if there are 20 CPU cores and the intensity is 30, we generate a total of 660 requests distributed uniformly in the 60-second window.

The average response time for the function selected uniformly from Table 6.1 is $\sim$ 1.042s. Thus, we state that for intensity 30, the CPU executes the function calls for roughly 50% of the time. Consequently, higher intensities correspond to higher actual utilization, and progressively more overloaded systems. Intensity 40 should make the processor execute the functions for 65% of time, and we additionally consider the intensity of 60 (and, in the appendix [140], intensities 90 and 120). Our estimations of CPU utilization do not take into account overheads of container creation and management. In fact, intensity 30 may result in full utilization of the processor if managing the container executing the function requires more time, on average per call, than executing the function itself.

Even with the same intensity, the number of CPU cores may still impact the performance. For example, some strategies may behave more flexibly when 660 requests are executed on 20 cores compared to when 330 requests are executed on 10 cores, although in both cases the intensity is equal to 30. With the increasing number of cores, we observe the effects of the regression to the mean utilization on a single core, i.e., the load can be balanced between cores more efficiently. For this reason, we perform our experiments on 10 and 20 cores. In the appendix [140], we also present the results on 5 cores.

For each pair of hyperparameters: numbers of CPUs and intensity, we generate 5 different random sequences of calls.

As the baseline, we use OpenWhisk with a single configuration change, the extended time limit. We set the same CPU and memory limits on the baseline workers as on our variants.

6.4 Cold starts, evictions and the impact on the response time

Theoretical considerations on scheduling policies often assume that the functions can be executed exactly when needed (cf. Chapter 5). Unfortunately, this is not always a case in practice. Although some requests are processed by idle containers that were already initialized (*warm start*), others may — and sometimes even must — be served by a new container (*cold start*). It takes 500 ms on the average [60] (and, in our measurements, up to 2 s) to fully initialize a new container, and thus the response time of a cold started request is always longer than a warm started one. On the other hand, reducing the total processing time by maintaining an unlimited number of containers is impossible due to limited memory, so it is profitable to evict (remove) unused containers.

Decisions on which containers to create and which to remove significantly impact the response time. However, these are orthogonal to what we measure in this paper — the impact of the scheduling policy on the response times of individual requests. With evictions, even small, quasi-random changes in the scheduling policies can be randomly amplified by evictions, which would introduce noise to our results. We decided to diminish this effect by reducing the number of evictions (and thus, cold starts) to zero or almost zero.

Our policies upper-bound the maximum number of containers in use by the number of functions and the CPU cores. Thus, we expect that with increasing memory capacity, the number of evictions will eventually approach zero, and thus the number of measured cold starts will be almost zero (as we pre-warm the containers before the experiment starts). On the other hand, Apache OpenWhisk is greedy: a pending request that has no free, warm container will trigger initialization of a new container. Thus, we expect that with increasing memory capacity, the number of cold starts will decrease, but only slowly.

We verified these intuitions experimentally. We measured the number of cold starts for different load intensities and for increasing memory (following the setup as in Section 6.3.1). The results are presented in Figure 6.2. For the original approach, Figure 6.2(a), the number of cold starts strongly depends on the intensity, but less so on the memory. In fact, for load intensity 120, the node processes 1 320 requests, and the number of cold starts exceeds 1 100 with almost no dependency on the amount of available memory. This means that over 80% of requests were processed on newly-created containers, each with the cold start overhead of almost 2 seconds. Moreover, for 128 GB of RAM, the number of concurrently-running containers was so large that Docker had problems running them — and OpenWhisk responded with a barrage of errors.

In contrast, in our FIFO policy, Figure 6.2(b), starting from 32 GB, the number of cold starts does not change — which suggests that RAM is no longer a constraint, and thus almost no evictions happen.

Our goal was to determine the memory threshold starting from which evictions can be neglected — and then run the rest of our experiments with this setup. Based on the above, we perform all further experiments with the OpenWhisk memory pool restricted to 32 GB.

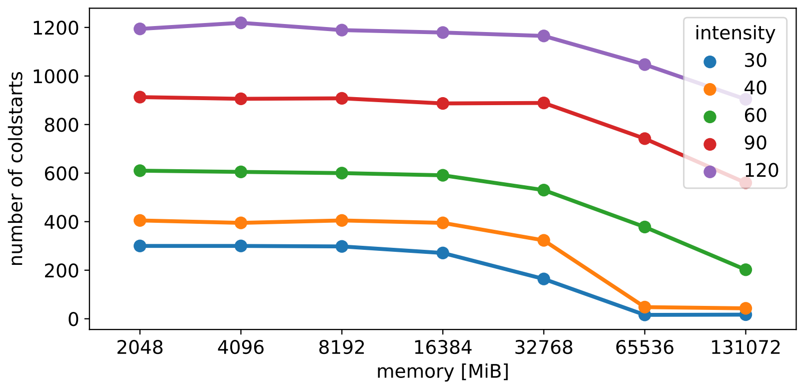
6.5 Influence of scheduling policies

We start with an aggregated view over all considered load intensities and CPU cores count (Section 6.5.1). We then analyze the influence of the increasing load intensities (Section 6.5.2) and CPU cores (Section 6.5.3). Finally, we change the relative call frequencies to highlight the fairness provided by the Fair-Choice (FC) strategy (Section 6.5.4). Results presented here and in the following section aggregate over 5 repetitions with different call sequences (the variance between repetitions is small); our on-line appendix [140] shows results for each of the 5 experiments.

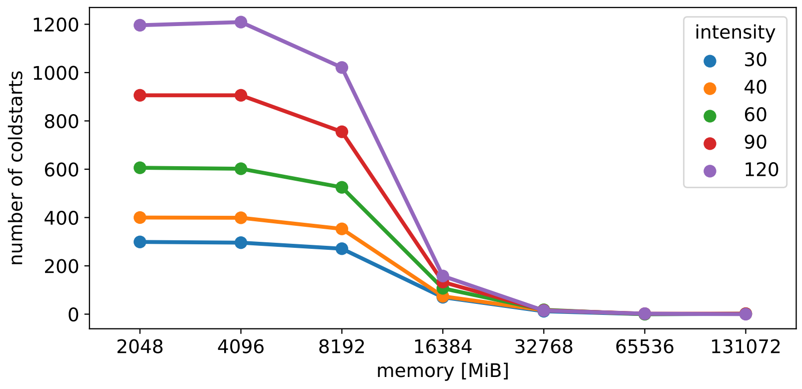
Figures 6.3 and 6.4 aggregate response times and stretches from all 5 experiments related to each combination of CPU cores and load intensity. Each row shows results for the same number of cores; each column — for the same intensity. To calculate stretch, we used the median response time of an idle system (Table 6.1). Thus, the stretch can be smaller than 1.

6.5.1 Influence of our strategies

We start by comparing the five proposed scheduling strategies against each other and against the baseline approach of OpenWhisk by computing the



(a) OpenWhisk baseline node-level scheduling

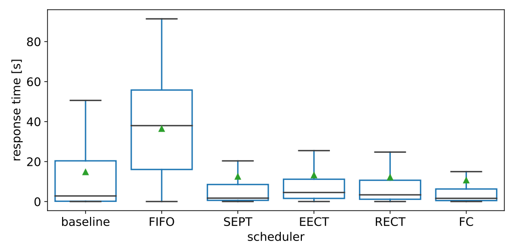


(b) our approach to node-level scheduling (FIFO variant)

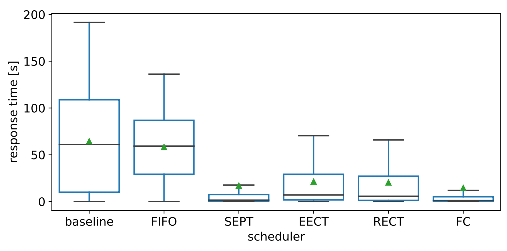
Figure 6.2: Comparison of the number of cold starts on 10 CPU cores depending on intensity and amount of available memory.

baseline-to-us ratio, i.e., the relative improvement of a given metric (response time, stretch or the completion time) that comes from applying a given strategy — and this over all considered CPU counts and load intensities (unless otherwise noted).

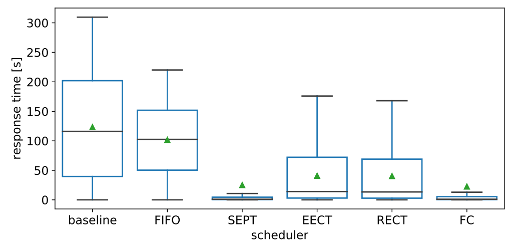
We start by analyzing our FIFO policy. In FIFO, the sequencing of calls is similar to the baseline OpenWhisk — but our FIFO effectively eliminates preemption (see Section 6.2.1 for details) and limits cold starts (Section 6.4). This impact can be best measured by comparing the time needed to process all the requests. Each test case processes exactly the same (base) load — thus, the differences in observed completion time are mostly caused by context switches and cold starts. Table 6.2 shows that our FIFO always reduces the completion time when there are 20 CPU cores involved; the improvement



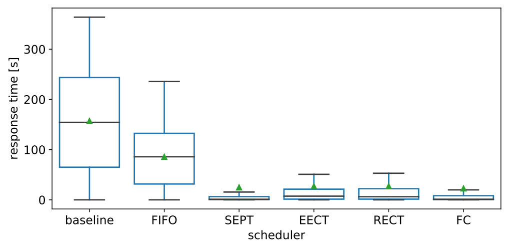
(a) 10 CPU cores, intensity 30



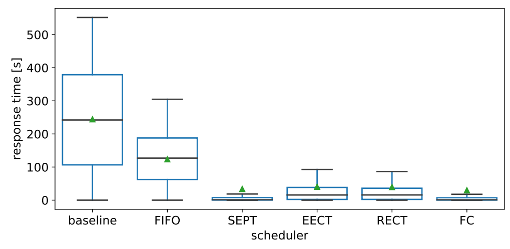
(b) 10 CPU cores, intensity 40



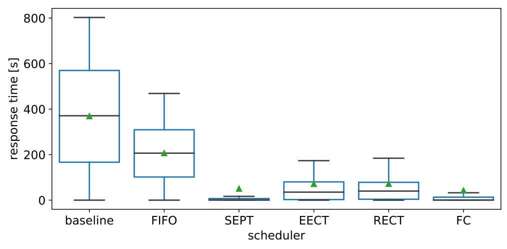
(c) 10 CPU cores, intensity 60



(d) 20 CPU cores, intensity 30



(e) 20 CPU cores, intensity 40

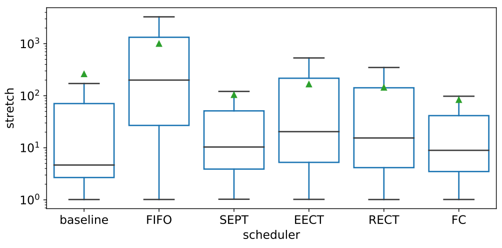


(f) 20 CPU cores, intensity 60

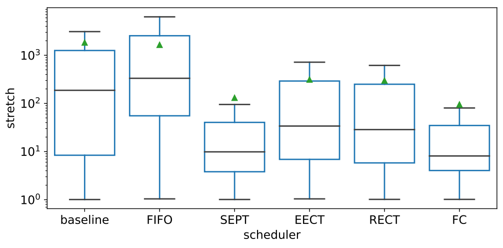
Figure 6.3: Response time for different scheduling policies, the number of CPU cores available for action containers (rows), and load intensity (columns). On-premise infrastructure. Here and in remaining plots, unless noted otherwise, each box aggregates results from all individual calls from all 5 sequences of calls. Thus, e.g., in (l), each box shows statistics over $5 \cdot 2640$ individual calls.

varies from 22% (intensity 30) to 45% (intensity 120). For 10 CPU cores, the improvement varies from -28% (which is actually a deterioration) to 34%.

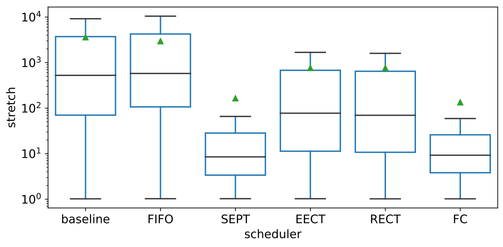
We continue by aggregating the response time statistics of Figures 6.3 and 6.4. Comparing FIFO to baseline, the average relative improvement of the average response time is 1.39. However, this improvement strongly depends on both the number of cores and load intensity, increasing with both parameters — with 10 CPUs, the improvement factor varies from 0.41



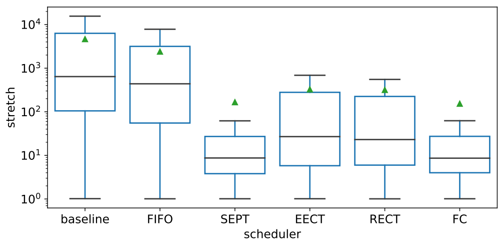
(a) 10 CPU cores, intensity 30



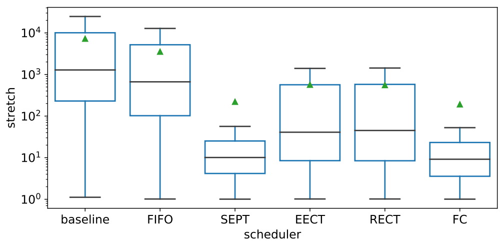
(b) 10 CPU cores, intensity 40



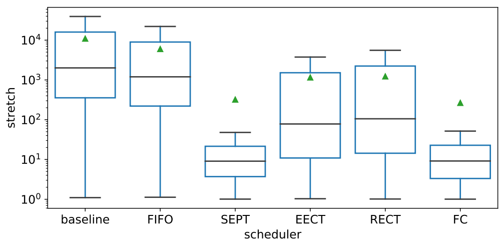
(c) 10 CPU cores, intensity 60



(d) 20 CPU cores, intensity 30



(e) 20 CPU cores, intensity 40



(f) 20 CPU cores, intensity 60

Figure 6.4: Stretch for different scheduling policies, the number of CPU cores available for action containers (rows), and load intensity (columns). On-premise infrastructure. Average stretch presented as a green triangle.

for intensity of 30 (i.e., an increase of the response time by the factor of over 2), to 1.21 for intensity of 60. With 20 CPUs, however, the improvement ranges from 1.79 to 1.98. For stretch, the relative improvements are almost the same (ranging from 0.26 to 1.22 for 10 CPUs, and from 1.82 to 2.05 for 20 CPUs). When it comes to the response time tails, these values are also decreased — the improvement factor for 10 CPUs varies between 1.02 and 1.41 for the 95th percentile of the response time, and between 0.98 and 1.44 for the 99th percentile. In case of 20 CPUs, these values range between 1.85 and 2.02, and 1.70 and 1.90, respectively.

Our remaining strategies show the additional benefit of smarter queuing. Overall, both SEPT and Fair-Choice improve upon FIFO in all experiments: the average relative response time improvement of SEPT is 3.59 and of FC is 4.10; while the average relative stretch improvement of SEPT is 14.89 and of FC is 18.02. Although one can see a relative degradation of the 99th percentile of response time compared to FIFO, there is still a visible improvement compared to the baseline: the average improvement factor for SEPT is 1.16, and FC is 1.40. EECT and RECT also show an advantage over FIFO (with the average relative response time improvement of 2.88). However, both these strategies prevent calls from starving.

6.5.2 Influence of the intensity

With the same number of CPU cores available for action containers, when load intensity increases, the queue of action calls waiting to be processed gets longer. Thus, the choice of the sequencing strategy is expected to have a more significant impact. We thus anticipate that the advantage of SEPT and FC over FIFO will increase with increasing load intensity.

Even with intensity 30, the advantage of SEPT and FC over FIFO, EECT and RECT can be clearly seen both for response time and for stretch. SEPT, FC, EECT and RECT perform better than FIFO in terms of the average response time (with the ratio of FIFO-to-other varying from 2.5 to 4.8) and the stretch (ratio from 3.9 to 22.6). Improvements further increase for intensities above 60 [140].

Consider the case of 10 CPU cores. For the intensity of 30, the average response time from our FIFO and SEPT strategies is 36.42 s and 12.52 s (with a ratio of 2.9). For intensity 40, these values are equal to 58.29 s and 17.01 s (with a ratio of 3.4); and for intensity 60, to 101.76 s and 25.14s (with a ratio of 4.0). At the same time, the FIFO-to-SEPT ratio of the median

Table 6.2: Maximum request completion times. For each pair of parameters (CPU cores/intensity) we show FIFO to baseline ratios (over 5 experiments). Configurations for which our method improves upon FIFO in each experiment in bold.

int. → cores ↓	30	40	60	90	120
5	1.14–1.20	1.10–1.13	0.98–1.05	0.97–1.02	**0.90–0.98**
10	1.11–1.28	**0.76–0.90**	**0.74–0.90**	0.92–1.04	**0.66–0.70**
20	**0.67–0.78**	**0.59–0.66**	**0.60–0.64**	**0.57–0.60**	**0.55–0.58**

response time varies from 22.0, through 38.7, to 95.9. This quick growth can be explained as follows: for large load intensity, we may almost freely choose short functions from the queue. This drastically reduces the median response time. Similarly, when it comes to stretch and intensity 30, the averages provided by our FIFO and SEPT strategies are 1000 and 104 (with a ratio of 9.6). For intensity 40, it is 1647 and 130 (with a ratio of 12.6), and for intensity 60, it is 2959 and 164 (with a ratio of 18.0), respectively. At the same time, the FIFO-to-SEPT ratio of median stretch increases from 19.3, through 33.7, to 68.0. This phenomenon can be explained in the same way.

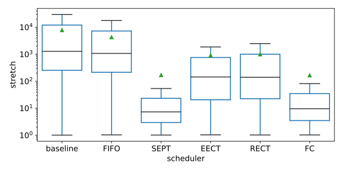
The significant influence of the scheduling strategy can be concluded from the fact that the difference between the baseline approach and our approaches remains stable in terms of the order of magnitude. For 20 CPU cores and for intensities 30, 40, 60, 90 and 120, the ratio of average response time obtained for the baseline and for our FIFO implementation is 1.8, 2.0, 1.8, 1.8 and 1.9, respectively. Similarly, the ratio of average stretch obtained for the baseline and for our FIFO implementation is 1.9, 2.0, 1.8, 1.8 and 1.9.

6.5.3 Influence of the number of CPU cores

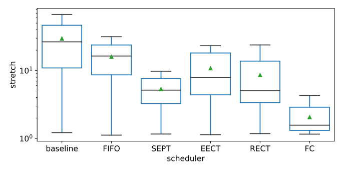
When the number of CPU cores available for action containers increases while the intensity remains the same, the load — although the same, on average, for each core — can be better balanced between cores. On the other hand, higher load is processed on more cores that share parts of their caches. Thus, we expect that with the increase of CPU cores, the improvements of our strategies will change slowly. On the other hand, we expect the baseline approach to behave worse for a higher number of CPU cores due to its container management policies. Our results confirm this intuition, especially for higher intensities.

Consider the intermediate load intensity 40. If 10 CPU cores are used, the average response times for our FIFO and SEPT implementations are 58.29 s and 17.01 s, with their ratio equal to 3.4. When the number of CPU is 20, however, the average values for FIFO and SEPT are 123.64s and 33.92s, with their ratio equal to 3.6. At the same time, the baseline-to-FIFO ratio of the average response time increases, from 1.1 to 2.0.

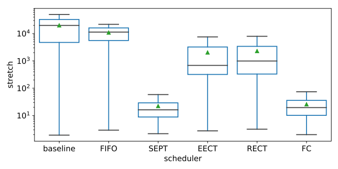
The same observations can be made for stretch. For 10 CPU cores, the average stretches for our FIFO and SEPT implementations are 1647.40 and 130.87, with their ratio equal to 12.6. When the number of CPU cores is 20, these values are 3538.65 and 220.89. As expected, the ratio of the average stretch for the baseline and for our FIFO implementation increases, from 1.1 to 2.0.



(a) All functions



(b) `dna-visualisation` (1% of all calls)



(c) `graph-bfs` (9.9% of all calls)

Figure 6.5: Comparison of aggregated stretch for different scheduling approaches and functions, 10 CPU cores and intensity 90. The `graph-bfs` function is short (12 ms) and thus we expect larger values of stretch compared to the long (8 552 ms) `dna-visualisation` function.

For FIFO and with fixed intensity (40 or more), the median response time (and thus, also stretch) increases almost linearly with the increase of the number of CPU cores (we confirmed that by analyzing additional results for 5 CPU cores [140]). If we increase the number of cores twice, the median response time and stretch will double. Our experiments thus show that the system overheads (related to container management) have a significant impact on the overall performance. At the same time, in the case of the baseline, doubling the number of cores triples the medians. Surprisingly, for the same core-level intensity, the best performance is presented by nodes that have lower numbers of cores.

6.5.4 Function-level metrics

The SEPT strategy will always prioritize short calls, independently of how often corresponding functions are called. As a consequence, if a long function is called relatively rare, it will be discriminated against. In Section 6.2, we proposed the Fair-Choice (FC) strategy which prioritizes calls based on the total resource consumption in a moving window. In order to quantify the fairness of the FC strategy, we performed 5 additional groups of experiments for the intermediate configuration with 10 CPU cores and intensity 90. In these experiments, exactly 10 calls corresponded to the relatively-long `dna-visualisation` function. Other calls were uniformly distributed among other functions, including a very short `graph-bfs` function. In contrast to our previous experiments, this time we made no assumptions on partial-uniformity of the call distribution. Figure 6.5 presents the aggregated stretch.

In Figure 6.5(a) we observe the distribution of stretch among all the calls. Although the call frequencies differ, Figure 6.5(a) is almost identical to Figure 6.4(c). The fairness of FC is apparent when we show just the results of the infrequent, long `dna-visualisation` function, Figure 6.5(b). FC reduces the average stretch from 5.3 (SEPT) to 2.1; the median stretch is reduced from 5.2 (SEPT) to 1.6, which hints that often the call is started almost immediately. These gains are not for free, though, Figure 6.5(c): FC increases the average stretch for a short, often-called `graph-bfs` function to 25.8 (compared with SEPT's 22.2).

6.6 Multiple worker nodes

Encouraged by our single-node results, we evaluate our approach in a multi-node environment. We measure the performance of a setup in which up to 4 workers operate in parallel. To do so, we created 5 virtual machines in our on-premises cloud running under Proxmox 7.1 [141] on physical machines equipped with AMD EPYC 7402P CPUs @ 2.80GHz with 24 hyper-threaded cores and 128 GB or RAM. Our controller is running on a VM limited to 4 CPU cores and 16 GB of RAM. Each of the four VMs executing OpenWhisk workers were assigned 20 CPU cores and 40 GB of RAM. The software setup of these workers is similar to the on-premises setup: two cores are reserved for system and invoker processes, while 18 were reserved for the action containers.

In each multi-node experiment, we send the same sequence of 2376 requests uniformly during a 60-second time-window. With 4 workers, 18-CPU nodes, 2376 corresponds to the core-level intensity of 30. We test the baseline and FC, but in different runs we reduce the number of available worker nodes from 4 up to 1. With 3 workers operating in parallel, 2376 requests correspond to the core-level intensity of 40; 2 workers correspond to the intensity of 60; and 1 worker to 120. As in the previous section, we repeated the experiment 5 times with different request sequences. Figure 6.6 shows the results.

With 3 machines, our FC strategy provides better quality of service than the baseline using 4 machines. For the baseline on 4 VMs, the average response time is 240 seconds (with the 75th percentile of 406 seconds, the 95th percentile of 600 seconds, and the 99th percentile of 649 seconds). Although our FC strategy processed the same load on just 3 VMs, we reduce the average response time to 68 seconds (71% less than the baseline on 4 VMs), its 75th percentile of 54 seconds (97% less), the 95th percentile of 443 seconds (26% less), and even the 99th percentile of 638 seconds (2% less).

Surprisingly, even on 2 VMs, some metrics are still better for our FC strategy compared to the baseline operating on 4 VMs. FC reduces the

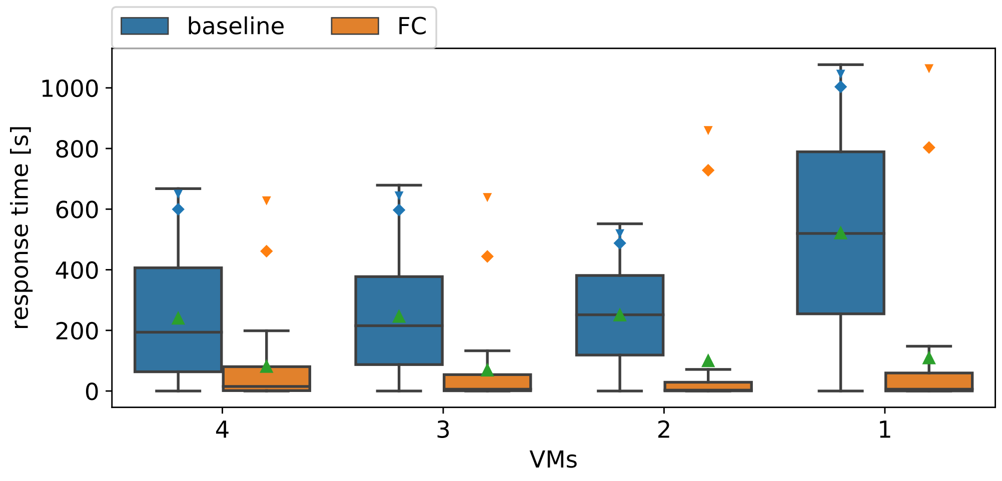


Figure 6.6: Response times in a multi-node environment. Each configuration processes the same sequence of 2376 requests. △ is the average; ◇ the 95th percentile; ▽ the 99th percentile.

average response time by 58% and its 75th percentile by 93%. Although FC worsens the 95th percentile by 21% and the 99th percentile by 32%, we argue that 95th percentile response time during a load peak corresponds to a much higher percentile in a longer time window.

6.7 Summary and discussion

In this chapter, we extended our methods of allocating computational resources on a single FaaS node from Chapter 5. Our aim was to reduce the response time and the stretch in a temporarily overloaded system. We modified Apache OpenWhisk and introduced a number of sequencing policies. We took advantage of the fact that in FaaS, each function is usually called repeatedly. This way, we were able to implement strategies that make use of historical data on the executions of a specific function. These strategies include adaptations of FIFO and SEPT, together with three new policies: EECT, RECT, and FC. In the case of these new policies, we made use of both the expected processing time of a function call and the frequency of calls of the same function in the recent past.

Compared to the baseline approach, the current version of OpenWhisk, our proposed algorithms show numerous advantages. For 20 CPUs our algorithms improve the average response time for all considered load intensities.

Compared to other analyzed strategies, SEPT and FC significantly decrease the response time — median, average, and the tail percentiles. At the same time, the FC strategy prioritizes jobs based on their previous usage, introducing inter-function fairness in the system. We also showed that all the response-time metrics for our implementation of the FC strategy are better on 3 VMs compared to the baseline running on 4 VMs. This means that our solution allows us to reduce the number of machines by a factor of at least 25% without decreasing the quality of the service.

Chapter 7

Conclusions

In this thesis, we analyzed a modern cloud computing model – *Function as a Service*. This model differs significantly from other cloud computing models by introducing the concept of a stateless *function* and thus freeing the cloud customers from the responsibility of maintaining cloud environments. At the same time, the properties of FaaS workloads open up new opportunities for optimization. We analyzed in detail three problems directly stemming from the FaaS computation model.

As we show in Chapter 3, the cloud provider can significantly optimize FaaS performance when they know the structure of the compositions used in the workload. Such knowledge enables the cloud provider to provision execution environments in advance and to improve scheduling. While we perform evaluation by simulations, presented algorithms could be implemented in FaaS systems and we discussed changes in the architecture that might be required. Our simulation results show that for non-negligible setup times (i.e., at least 20 times longer than the average task duration) in all evaluated machine configurations when the scheduling algorithm is aware of task dependencies and startup times, the average improvement of the response latency is at least by the factor of two. In practice, setup times can take hundreds of milliseconds [60], while the function may complete processing within several milliseconds [54].

In Chapter 4, we analyzed a two-dimensional resource management problem with applications having multiple instances. The analysed model maps to cloud applications where particular instances (function environments, Docker containers, Virtual Machines) can be replicated, and each instance is assigned the same amount of resources (memory). Then, incoming requests can be balanced between multiple instances. Such an approach integrates separate components: the scheduler, the autoscaler, and the load balancer. We present a number of theoretical results and polynomial algorithms that solve special cases with applications having equal requirements. For the general case, we propose heuristics and identify the cases where we significantly reduce the number of used machines, often achieving the lower bound.

Nevertheless, the characteristics of future load often cannot be predicted precisely – in particular, we are not aware of task release times, and after the tasks are scheduled it may occur that some of them would use more resources (e.g., CPU time) than expected. Thus, in Chapter 5, we studied various non-clairvoyant scheduling strategies for a single node in a large FaaS cluster. We focused on improving end-user experience, thus we analyzed metrics related to response time or stretch. Our heuristics rely on information gathered locally on a single node, such as past call durations and frequency of the calls. We analyzed both preemptive and non-preemptive variants of our algorithms. In both cases, we observed significant improvement over the baselines. In particular, SERPT (in the preemptive variant) reduces the average flow time by a factor of 1.4 and SEPT (in the non-preemptive variant) reduces the average flow time by a factor of 6. We also indicate that our newly introduced Fair Choice strategies, which outperform other algorithms when measuring the flow time for function-aggregated metrics, can be effectively applied to the FaaS system.

We further investigated this approach from the system perspective. In Chapter 6, we adapt methods of allocating computational resources on a single FaaS node introduced in Chapter 5. We modified Apache OpenWhisk and we implemented strategies that make use of historical data of the executions of a specific function. These strategies include adaptations of policies analyzed in Chapter 5 together with the new policies: EECT and RECT. We extended OpenWhisk implementation to record information about function calls and provide estimations used in proposed strategies. We show that the proposed algorithms provide the largest improvements over the current version of OpenWhisk in scenarios with the highest request intensities and a large number of CPUs. In particular, SEPT and FC policies provide a significant decrease in response time. Moreover, FC strategy makes fair decisions based on the total CPU consumption of functions in the past. These decisions prioritize longer functions that are called rarely and decrease their stretch when compared to shorter yet regularly-called functions.

Improvements analyzed in this thesis do not require significant modifications in the FaaS cloud architecture. As we do not change the interfaces for declaring and managing the FaaS workload, our proposed solutions could be adapted without requiring any modifications in deployed applications. There is also no need to perform changes in the data center infrastructure, thus our proposed changes do not require any (potentially costly) modifications in the cloud provider's internal hardware stack. Moreover, as some of the proposed solutions require changes only on the worker node management software, they can be introduced in a rolling scheme, allowing to verify the correctness of the setup and measuring benefits without risking the stability of the whole system. In addition, the proposed methods can be combined with each other to improve overall performance.

With FaaS, the cloud provider is able to gain more information about executed applications – as functions are short and executed in a repeatable manner, it is possible to estimate the requirements of the functions. Moreover, cloud users can provide more information about FaaS workloads, such as limits, dependencies or the structure of composite FaaS functions. Our analysis of the FaaS model indicates that characteristics of the FaaS workload enable the cloud provider to perform more accurate scheduling decisions compared to applications built and deployed with other mature technologies, such as containers or Virtual Machines. We believe that with the increasing popularity of serverless computing, scheduling methods designed to benefit from FaaS attributes will play an important role in modern cloud computing platforms.

Bibliography

[1] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation", in *Proceedings of the 15th European conference on computer systems (EuroSys)*, ACM, 2020.

[2] *Okeanos*, ICM UW. [Online]. Available: `https://kdm.icm.edu.pl/Zasoby/komputery_w_icm.pl/` (visited on 2022-12-07).

[3] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks", *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[4] S. G. Langer and T. French, "Virtual Machine Performance Benchmarking", *Journal of Digital Imaging*, vol. 24, no. 5, pp. 883–889, 2011.

[5] *Kubernetes*. [Online]. Available: `https://kubernetes.io` (visited on 2022-11-08).

[6] *Docker Swarm mode overview*. [Online]. Available: `https://docs.docker.com/engine/swarm/` (visited on 2022-11-08).

[7] P. Mell, T. Grance, *et al.*, "The NIST Definition of Cloud Computing", National Institute of Standards and Technology, 2011. DOI: `10.6028/NIST.SP.800-145`.

[8] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing", *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019. DOI: `10.1145/3368454`.

[9] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems", in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, and R. Buyya, Eds. Singapore: Springer Singapore, 2017.

[10] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing", *arXiv preprint arXiv:1902.03383*, 2019.

[11] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "RFaaS: RDMA-Enabled FaaS Platform for Serverless High-Performance Computing", *arXiv:2106.13859 [cs]*,

[12] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures", in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, ACM, 2018. DOI: `10.1145/3185768.3186308`.

[13] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing", *Journal of Cloud Computing*, vol. 10, no. 1, p. 39, 2021-07. DOI: `10.1186/s13677-021-00253-7`.

[14] S. Pedratscher, S. Ristov, and T. Fahringer, "M2FaaS: Transparent and fault tolerant FaaSification of Node.js monolith code blocks", *Future Generation Computer Systems*, vol. 135, pp. 57–71, 2022-10, ISSN: 0167-739X. DOI: `10.1016/j.future.2022.04.021`.

[15] M. Copik, A. Calotoiu, K. Taranov, and T. Hoefler, "FaasKeeper: A Blueprint for Serverless Services", *arXiv preprint arXiv:2203.14859*, 2022.

[16] *Apache ZooKeeper*, Apache Software Foundation. [Online]. Available: `https://zookeeper.apache.org/` (visited on 2022-11-08).

[17] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A Fault-Tolerance Shim for Serverless Computing", in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, ACM, 2020-04, pp. 1–15. DOI: `10.1145/3342195.3387535`.

[18] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized Core-Granular Scheduling for Serverless Functions", in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, Santa Cruz, CA, USA: ACM, 2019, pp. 158–164. DOI: `10.1145/3357223.3362709`.

[19] *Openwhisk documentation*, Apache Software Foundation. [Online]. Available: `https://openwhisk.apache.org/documentation.html` (visited on 2021-05-17).

[20] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, "FaaSRank: Learning to Schedule Functions in Serverless Platforms", in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, IEEE, 2021, pp. 31–40.

[21] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling Resource Underutilization in the Serverless Era", in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20, ACM, 2020, pp. 280–295. DOI: `10.1145/3423211.3425683`.

[22] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing", in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, ACM, 2020, pp. 1–15. DOI: `10.1145/3419111.3421286`.

[23] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service Workflows", in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 805–820.

[24] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting", in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, ACM, 2020, pp. 467–481. DOI: `10.1145/3373376.3378512`.

[25] Google, *Google gVisior.* [Online]. Available: `https://gvisor.dev/` (visited on 2021-09-30).

[26] S. Thomas, L. Ao, G. M. Voelker, and G. Porter, "Particle: Ephemeral Endpoints for Serverless Networking", in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, ACM, 2020, pp. 16–29. DOI: `10.1145/3419111.3421275`.

[27] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip Redundant Paths to Make Serverless Fast", in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, ACM, 2020-04, pp. 1–15. DOI: `10.1145/3342195.3392698`.

[28] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, and C. Abad, "Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms", in *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 282–291. DOI: `10.1109/CCGRID.2019.00042`.

[29] Z. Li, Q. Chen, and M. Guo, "Pagurus: Eliminating Cold Startup in Serverless Computing with Inter-Action Container Sharing", *arXiv:2108.11240*,

[30] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency", in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, IEEE, 2021, pp. 797–803.

[31] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, “Pushing Serverless to the Edge with WebAssembly Runtimes”, in *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 140–149. DOI: `10.1109/CCGrid54584.2022.00023`.

[32] D. Bermbach, A.-S. Karakaya, and S. Buchholz, “Using Application Knowledge to Reduce Cold Starts in FaaS Services”, in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ACM, 2020, pp. 134–143. DOI: `10.1145/3341105.3373909`.

[33] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet”, in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20, ACM, 2020, pp. 45–59. DOI: `10.1145/3419111.3421297`.

[34] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, “OFC: An Opportunistic Caching System for FaaS Platforms”, in *Proceedings of the Sixteenth European Conference on Computer Systems*, Online Event United Kingdom: ACM, 2021-04, pp. 228–244. DOI: `10.1145/3447786.3456239`.

[35] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, “SAND: Towards High-Performance Serverless Computing”, in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 923–935.

[36] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, “Cloudburst: Stateful Functions-as-a-Service”, *arXiv preprint arXiv:2001.04592v2*, 2020-01.

[37] A. Banaei and M. Sharifi, “ETAS: Predictive scheduling of functions on worker nodes of Apache OpenWhisk platform”, *The Journal of Supercomputing*, 2021. DOI: `10.1007/s11227-021-04057-z`.

[38] Y. Fu, L. Liu, H. Wang, Y. Cheng, and S. Chen, “SFS: Smart OS Scheduling for Serverless Functions”, *arXiv preprint arXiv:2209.01709*, 2022.

[39] S. Martello and P. Toth, “Bin-packing problem”, *Knapsack problems: Algorithms and computer implementations*, pp. 221–245, 1990.

[40] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, “Approximation and online algorithms for multidimensional bin packing: A survey”, *Computer Science Review*, vol. 24, pp. 63–79, 2017.

[41] S. S. Seiden, “On the Online Bin Packing Problem”, *Journal of the ACM (JACM)*, vol. 49, no. 5, pp. 640–671, 2002.

[42] E. G. Coffman Jr, K. So, M. Hofri, and A. Yao, "A Stochastic Model of Bin-Packing", *Information and control*, vol. 44, no. 2, pp. 105–115, 1980.

[43] M. Drozdowski, *Scheduling for Parallel Processing.* Springer, 2009.

[44] G. Centeno and R. L. Armacost, "Parallel machine scheduling with release time and machine eligibility restrictions", *Computers and Industrial Engineering*, vol. 33, no. 1, pp. 273–276, 1997, Proceedings of the 21st International Conference on Computers and Industrial Engineering. DOI: 10.1016/S0360-8352(97)00091-0.

[45] A. Allahverdi, J. N. Gupta, and T. Aldowaisan, "A review of scheduling research involving setup considerations", *Omega*, vol. 27, no. 2, pp. 219–239, 1999.

[46] D. B. Shmoys, J. Wein, and D. P. Williamson, "Scheduling parallel machines on-line", *SIAM Journal on Computing*, vol. 24, no. 6, pp. 1313–1331, 1995.

[47] J. K. Lenstra and A. Rinnooy Kan, "Complexity of Scheduling under Precedence Constraints", *Operations Research*, vol. 26, no. 1, pp. 22–35, 1978.

[48] P. Brucker, *Scheduling Algorithms.* Springer, 2007. DOI: 10.1007/978-3-540-69516-5.

[49] A. Allahverdi, C. T. Ng, T. E. Cheng, and M. Y. Kovalyov, "A survey of scheduling problems with setup times or costs", *European Journal of Operational Research*, vol. 187, no. 3, pp. 985–1032, 2008.

[50] V. Bharadwaj, D. Ghose, and T. G. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems", *Cluster Computing*, vol. 6, no. 1, pp. 7–17, 2003.

[51] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at Google", in *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, ACM, 2020. DOI: 10.1145/3342195.3387524.

[52] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, *et al.*, "Ananta: Cloud Scale Load Balancing", *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013.

[53] L. A. Barroso, J. Clidaras, and U. Hölzle, “The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines”, *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[54] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, “SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing”, in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware 2021, ACM, 2021. DOI: `10.1145/3464298.3476133`.

[60] M. Shahrad, J. Balkind, and D. Wentzlaff, “Architectural Implications of Function-as-a-Service Computing”, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2019, pp. 1063–1075. DOI: `10.1145/3352460.3358296`.

[61] B. Gacias, C. Artigues, and P. Lopez, “Parallel machine scheduling with precedence constraints and setup times”, *Computers & Operations Research*, vol. 37, no. 12, 2010.

[62] M. Afzalirad and J. Rezaeian, “Resource-constrained unrelated parallel machine scheduling problem with sequence dependent setup times, precedence constraints and machine eligibility restrictions”, *Computers & Industrial Engineering*, vol. 98, 2016.

[63] *Apache OpenWhisk*, Apache Software Foundation, 2020. [Online]. Available: `https://openwhisk.apache.org`.

[64] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Serverless Computation with OpenLambda”, in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, USENIX Association, 2016-06.

[65] K. Kritikos and P. Skrzypek, “A review of Serverless Frameworks”, in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 161–168. DOI: `10.1109/UCC-Companion.2018.00051`.

[66] *Scala documentation*, École Polytechnique Fédérale. [Online]. Available: `https://docs.scala-lang.org/` (visited on 2021-05-17).

[67] D. Wyatt, *Akka concurrency.* Artima Incorporation, 2013.

[68] *OpenWhisk Scheduling Proposal.* [Online]. Available: `https://cwiki.apache.org/confluence/download/attachments/85466849/OW_new_scheduling_proposal_180525.pdf` (visited on 2021-06-24).

[69] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The Serverless Trilemma: Function Composition for Serverless Computing", in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ACM, 2017.

[70] C. Chekuri and S. Khanna, "On Multi-dimensional Packing Problems", in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1999.

[71] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg", in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, ACM, 2015.

[72] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, 1979.

[73] H. Zhao and R. Sakellariou, "An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm", in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., Springer, 2003, pp. 189–194, ISBN: 978-3-540-45209-6.

[74] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira, "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm", in *18th Euromicro conference on parallel, distributed and network-based processing (PDP)*, IEEE, 2010.

[75] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider", in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, 2020, pp. 205–218.

[76] L. Wasserman, *All of Statistics: A Concise Course in Statistical Inference.* Springer, 2004, vol. 26.

[77] A. Hughes and D. Grawoig, *Statistics: A Foundation for Analysis.* 1971.

[78] A. Allahverdi, "The third comprehensive survey on scheduling problems with setup times/costs", *European Journal of Operational Research*, vol. 246, no. 2, 2015.

[79] *Gurobi Optimizer*, Gurobi Optimization, LLC. [Online]. Available: `https://www.gurobi.com/solutions/gurobi-optimizer/` (visited on 2023-01-23).

[80] M. X. Weng, J. Lu, and H. Ren, "Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective", *International Journal of Production Economics*, vol. 70, no. 3, 2001.

[81] S. Webster and M. Azizoglu, "Dynamic programming algorithms for scheduling parallel machines with family setup times", *Computers & Operations Research*, vol. 28, no. 2, 2001.

[82] F. Wu, Q. Wu, and Y. Tan, "Workflow scheduling in cloud: a survey", *The Journal of Supercomputing*, vol. 71, no. 9, 2015.

[83] A. Ilyushkin and D. Epema, "The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows", in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018.

[84] W. Shao, F. Xu, L. Chen, H. Zheng, and F. Liu, "Stage delay scheduling: Speeding up dag-style data analytics jobs with resource interleaving", in *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, 2019.

[85] R. Sakellariou and H. Zhao, "A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems", in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2004.

[86] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking Functions to Warm the Serverless Cold Start", in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20, ACM, 2020, pp. 1–13. DOI: `10.1145/3423211.3425682`.

[87] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing", in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, 2020-07, pp. 419–433, ISBN: 978-1-939133-14-4.

[88] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments", in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, IEEE, 2020, pp. 1–10.

[89] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, 2020-11, pp. 1187–1204, ISBN: 978-1-939133-19-9.

[90] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms", in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, ACM, 2017, pp. 153–167. DOI: 10.1145/3132747.3132772.

[91] P. Minet, E. Renault, I. Khoufi, and S. Boumerdassi, "Analyzing traces from a Google data center", in *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*, IEEE, 2018, pp. 1167–1172.

[92] R. McNaughton, "Scheduling with Deadlines and Loss Functions", *Management Science*, 1959, ISSN: 00251909, 15265501. DOI: 10.1287/mnsc.6.1.1.

[93] B. Przybylski, P. Żuk, and K. Rzadca, *Divide (CPU Load) and Conquer: Semi-Flexible Cloud Resource Allocation. Appendix.* [Online]. Available: https://www.mimuw.edu.pl/~krzadca/opal/data/bin-dl-appendix.pdf (visited on 2022-12-01).

[94] I. Pietri and R. Sakellariou, "Mapping Virtual Machines onto Physical Machines in Cloud Computing: A survey", *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, 2016, ISSN: 0360-0300. DOI: 10.1145/2983575.

[95] K. Fleszar and K. S. Hindi, "New heuristics for one-dimensional bin-packing", *Computers & Operations Research*, 2002, ISSN: 0305-0548. DOI: 10.1016/S0305-0548(00)00082-4.

[96] D. Castro-Silva and E. Gourdin, "A study on load-balanced variants of the bin packing problem", *Discrete Applied Mathematics*, vol. 264, pp. 4–14, 2019. DOI: 10.1016/j.dam.2018.07.010.

[97] A. Grange, I. Kacem, and S. Martin, "Algorithms for the bin packing problem with overlapping items", *Computers & Industrial Engineering*, 2018, ISSN: 0360-8352. DOI: 10.1016/j.cie.2017.10.015.

[98] I. Morihara, T. Ibaraki, and T. Hasegawa, "Bin packing and multiprocessor scheduling problems with side constraint on job types", *Discrete Applied Mathematics*, 1983, ISSN: 0166-218X. DOI: 10.1016/0166-218X(83)90071-9.

[99] S. Kamali, "Efficient Bin Packing Algorithms for Resource Provisioning in the Cloud", in *Algorithmic Aspects of Cloud Computing: First International Workshop (ALGOCLOUD)*, Springer, 2015. DOI: 10.1007/978-3-319-29919-8_7.

[100] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive Resource Provisioning for the Cloud Using Online Bin Packing", *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2647–2660, 2014. DOI: 10.1109/TC.2013.148.

[101] X. Tang, Y. Li, R. Ren, and W. Cai, "On First Fit Bin Packing for Online Cloud Server Allocation", in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016. DOI: 10.1109/IPDPS.2016.42.

[102] R. Ren, X. Tang, Y. Li, and W. Cai, "Competitiveness of Dynamic Bin Packing for Online Cloud Server Allocation", *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1324–1331, 2017. DOI: 10.1109/TNET.2016.2630052.

[103] K. Psychasand and J. Ghaderi, "High-Throughput Bin Packing: Scheduling Jobs With Random Resource Demands in Clusters", *IEEE/ACM Transactions on Networking*, vol. 29, no. 1, pp. 220–233, 2021. DOI: 10.1109/TNET.2020.3034022.

[104] W. Wei, K. Wang, K. Wang, S. Guo, and H. Gu, "A Virtual Machine Placement Algorithm Combining NSGA-II and Bin-Packing Heuristic", in *Proceedings of the 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2019. DOI: 10.1109/PDCAT46702.2019.00044.

[105] A. Fatima, N. Javaid, T. Sultana, M. Y. Aalsalem, S. Shabbir, and Durr-e-Adan, "An Efficient Virtual Machine Placement via Bin Packing in Cloud Data Centers", in *Advanced Information Networking and Applications*, Springer, 2019. DOI: 10.1007/978-3-030-15032-7_82.

[106] L. Eyraud-Dubois and H. Larchevêque, "Optimizing Resource allocation while handling SLA violations in Cloud Computing platforms", in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013. DOI: 10.1109/IPDPS.2013.67.

[107] F. F. Moges and S. L. Abebe, "Energy-aware VM placement algorithms for the OpenStack Neat consolidation framework", *Journal of Cloud Computing*, vol. 8, no. 1, pp. 1–14, 2019.

[108] N. Buchbinder, Y. Fairstein, K. Mellou, I. Menache, and J. S. Naor, “Online Virtual Machine Allocation with Lifetime and Load Predictions”, in *Abstract Proceedings of the 2021 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, ACM, 2021, pp. 9–10.

[109] S. Rampersaud and D. Grosu, “Sharing-Aware Online Virtual Machine Packing in Heterogeneous Resource Clouds”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 2046–2059, 2017. DOI: `10.1109/TPDS.2016.2641937`.

[110] O. Beaumont, L. Eyraud-Dubois, C. Thraves Caro, and H. Rejeb, “Heterogeneous Resource Allocation under Degree Constraints”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 926–937, 2013. DOI: `10.1109/TPDS.2012.175`.

[111] F. Chung, R. Graham, J. Mao, and G. Varghese, “Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines”, *Theory of Computing Systems*, vol. 39, no. 6, pp. 829–849, 2006-09. DOI: `10.1007/s00224-006-1249-3`.

[112] L. Epstein and R. van Stee, “Improved Results for a Memory Allocation Problem”, *Theory of Computing Systems*, vol. 48, no. 1, pp. 79–92, 2009. DOI: `10.1007/s00224-009-9226-2`.

[113] L. Epstein and R. van Stee, “Approximation schemes for packing splittable items with cardinality constraints”, in *Approximation and Online Algorithms*, C. Kaklamanis and M. Skutella, Eds., Springer, 2008, pp. 232–245, ISBN: 978-3-540-77918-6.

[114] O. Beaumont, J.-A. Lorenzo, L. Eyraud-Dubois, and P. Renaud-Goud, “Efficient and Robust Allocation Algorithms in Clouds under Memory Constraints”, in *Proceedings of the 21st International Conference on High Performance Computing (HiPC)*, 2014. DOI: `10.1109/HiPC.2014.7116894`.

[115] J. Ding, R. Cao, I. Saravanan, N. Morris, and C. Stewart, “Characterizing Service Level Objectives for Cloud Services: Realities and Myths”, in *2019 IEEE International Conference on Autonomic Computing (ICAC)*, 2019, pp. 200–206. DOI: `10.1109/ICAC.2019.00032`.

[116] M. Mao and M. Humphrey, “A Performance Study on the VM Startup Time in the Cloud”, in *2012 IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 423–430. DOI: `10.1109/CLOUD.2012.103`.

[117] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and Stretch Metrics for Scheduling Continuous Job Streams", in *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1998, pp. 270–279.

[118] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke, "Online Scheduling to Minimize Average Stretch", in *40th Annual Symposium on Foundations of Computer Science*, 1999. DOI: `10.1109/SFFCS.1999.814615`.

[119] N. Megow and A. S. Schulz, "On-line scheduling to minimize average completion time revisited", *Operations Research Letters*, vol. 32, no. 5, 2004, ISSN: 0167-6377. DOI: `10.1016/j.orl.2003.11.008`.

[120] A. Pérez, S. Risco, D. M. Naranjo, M. Caballer, and G. Moltó, "On-Premises Serverless Computing for Event-Driven Data Processing Applications", in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 414–421. DOI: `10.1109/CLOUD.2019.00073`.

[121] K. Baker, *Introduction to Sequencing and Scheduling.* Wiley, 1974, ISBN: 978-0-471-04555-7.

[122] B. T. Sloss, S. Nukala, and V. Rau, "Metrics that matter", *Communications of the ACM*, vol. 62, no. 4, pp. 88–94, 2019, ISSN: 0001-0782. DOI: `10.1145/3303874`.

[123] O. Bellenguez-Morineau, M. Chrobak, C. Dürr, and D. Prot, "A note on NP-hardness of preemptive mean flow-time scheduling for parallel machines", *Journal of Scheduling*, vol. 18, no. 3, 2015.

[124] J. Lenstra, A. Rinnooy Kan, and P. Brucker, "Complexity of Machine Scheduling Problems", *Annals of discrete mathematics*, vol. 1, 1977.

[125] A. Legrand, A. Su, and F. Vivien, "Minimizing the stretch when scheduling flows of divisible requests", *Journal of Scheduling*, vol. 11, no. 5, 2008. DOI: `10.1007/s10951-008-0078-4`.

[126] J. Bruno, E. Coffman Jr., and R. Sethi, "Scheduling Independent Tasks To Reduce Mean Finishing Time", *Communications of the ACM*, vol. 17, 1974.

[127] P. Baptiste, P. Brucker, M. Chrobak, C. Dürr, S. A. Kravchenko, and F. Sourd, "The complexity of mean flow time scheduling problems with release times", *Journal of Scheduling*, vol. 10, no. 2, 2007. DOI: `10.1007/s10951-006-0006-4`.

[128] J. A. Hoogeveen and A. P. A. Vestjens, "Optimal on-line algorithms for single-machine scheduling", in *Integer Programming and Combinatorial Optimization*, Springer, 1996, ISBN: 978-3-540-68453-4.

[129] P. Liu and X. Lu, "On-line scheduling of parallel machines to minimize total completion times", *Computers & Operations Research*, vol. 36, no. 9, 2009. DOI: 10.1016/j.cor.2008.11.008.

[130] J. R. Correa and M. R. Wagner, "LP-based online scheduling: From single to parallel machines", in *Integer Programming and Combinatorial Optimization*, Springer, 2005, ISBN: 978-3-540-32102-6.

[131] S. Leonardi and D. Raz, "Approximating total flow time on parallel machines", *Journal of Computer and System Sciences*, vol. 73, no. 6, 2007, ISSN: 0022-0000. DOI: 10.1016/j.jcss.2006.10.018.

[132] M. A. Bender, S. Muthukrishnan, and R. Rajaraman, "Approximation Algorithms for Average Stretch Scheduling", *Journal of Scheduling*, vol. 7, no. 3, 2004. DOI: 10.1023/b:josh.0000019681.52701.8b.

[133] N. Megow, M. Uetz, and T. Vredeveld, "Stochastic Online Scheduling on Parallel Machines", in *Approximation and Online Algorithms*, Springer, 2005, ISBN: 978-3-540-31833-0.

[134] N. Megow, M. Uetz, and T. Vredeveld, "Models and Algorithms for Stochastic Online Scheduling", *Mathematics of Operations Research*, vol. 31, no. 3, 2006. DOI: 10.1287/moor.1060.0201.

[135] T. Vredeveld, "Stochastic online scheduling", *Computer Science - Research and Development*, vol. 27, no. 3, 2011. DOI: 10.1007/s00450-011-0153-5.

[136] N. Megow and T. Vredeveld, "Approximation in Preemptive Stochastic Online Scheduling", in *Algorithms – ESA 2006, 14th Annual European Symposium*, Springer, 2006, ISBN: 978-3-540-38876-0.

[137] N. Megow and T. Vredeveld, "A Tight 2-Approximation for Preemptive Stochastic Scheduling", *Mathematics of Operations Research*, vol. 39, no. 4, 2014. DOI: 10.1287/moor.2014.0653.

[138] Gatling Corp, *Gatling Open-Source Load Testing*. [Online]. Available: https://gatling.io (visited on 2021-07-27).

[139] Apache Software Foundation, *Apache JMeter™*. [Online]. Available: https://jmeter.apache.org/ (visited on 2021-08-24).

[140] P. Żuk, B. Przybylski, and K. Rzadca, "Call Scheduling to Reduce Response Time of a FaaS System. Appendix.", *arXiv preprint arXiv:2207.13168*, 2022.

[141] *Proxmox VE*. [Online]. Available: https://pve.proxmox.com (visited on 2022-05-12).