CASY: A CPU Cache Allocation System for FaaS Platform

Armel Jeatsa-Toulepi, Boris Teabe, Daniel Hagimont

CASY: a CPU Cache Allocation System for FaaS Platform

Armel Jeatsa, Boris Teabe, Daniel Hagimont
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France

***Abstract*—Function as a Service (FaaS) has become a key service in the cloud. It enables customers to conceive their application as a collection of minimal serverless functions interacting with each other. FaaS platforms abstract all the management complexity to the client. This emerging paradigm is also attractive because of its billing model. Clients are charged based on the execution time of functions, allowing finer-grained pricing. Therefore, executing functions as fast as possible is very important to lower the cost.**

Several research studies have investigated runtime optimization in FaaS environments, but none have explored CPU cache allocation. Indeed, CPU cache contention is a well-known issue in software and FaaS is not exempt from this issue. Various hardware improvements have been made to address the CPU cache partitioning problem. Among other things, Intel has implemented a new technology in their new processors that allows cache partitioning: *Cache Allocation Technology* (CAT). This technology allows allocating cache ways to processes, and the usage of the cache by each process will be limited to the allocated amount.

In this paper, we propose *CASY* (CPU Cache Allocation SYstem), a system that performs CPU cache allocation for serverless functions using the Intel CAT technology. *CASY* uses machine learning to build a cache usage profile for functions and uses this profile to predict the cache requirements based on the function's input data. Because the CPU's cache size is small, *CASY* integrates an allocation algorithm which ensures that the cache loads are balanced on all cache ways. We implemented our system and integrated it into the OpenWhisk FaaS platform. Our evaluations show a 11% decrease in execution time for some serverless functions without degrading the performance of other functions.

***Index Terms*—FaaS, CPU Cache, Allocation**

I. Introduction

Today, with the advent of cloud computing [1], we are witnessing the emergence of new services such as Function As A Service [2] (*FaaS* for short). The FaaS model allows developers to create, run, and manage their applications as a set of stateless functions, without having to maintain their own infrastructure. The way FaaS work is as following: (1) The developer uploads their function written in a programming language supported by the FaaS platform (e.g. Java, Python, JavaScript, etc.) which are then saved in a registry (the registry is a kind of data base that saves the functions); (2) The developer specifies a set of events (e.g. new records in a database, reception of a specific request) that will trigger the execution of the functions; (3) Each time an event happens, the FaaS platform executes the corresponding function in a sandbox, which can be a container such as Docker [3] or a light virtual machine such as Firecracker [4]. There are several FaaS platforms than can be either supplied by Cloud providers (e.g google Cloud Functions [5]) or provided with a software that can be deployed to adminstrate the execution of function (e.g OpenWhisk [6]). In FaaS, most functions follow the Extract-Transform-Load (ETL) [7] execution model, which can be described as follows: (E) the function read the input data (e.g an image, a sound record) from persistent storage; (T) the function carries out computations on the data (e.g image processing, audio encoding); and finally, (L) the function output is stored in persistent storage. The ETL model motivates the use of sandboxes for function executions in order to have an isolated environment that contains all necessary dependencies. Besides the serverless aspect of FaaS, its attractiveness also comes from its billing system. Generally, with other types of cloud service (IaaS, PaaS and SaaS), the billing is done based on the execution time of the environment where the application is executed (e.g. total runtime of the virtual machine), while with FaaS the billing is done on the execution time of the function.

In fact, the billing in FaaS is done at each function invocation and is based on the total execution time [8], thus providing a fine-grained billing system. A shorter execution time for a function implies less cost. Therefore, execution time is a key element in FaaS. Several research works investigated approaches to reduce the execution time of functions. Many of these works are interested in the execution environment, such as optimization of container boot times [9], [10], whereas others proposed elaborated cache systems for data to shorten the loading step (the *E* step in ETL) before function execution [7], [11]. In this work, we are interested in optimizing the CPU cache in the context of FaaS execution.

Most CPUs use caches to speed up memory access. CPU caches are fast memory components close to a CPU core containing frequently-used memory entries [12]. Upon accessing a memory location, the CPU first checks for its presence in the cache and retrieves the data directly if present. Otherwise, the CPU would have to request this data from the main memory. To avoid cache misses, most modern CPUs use a hierarchy of multiple caches. Cache levels further from the CPU (e.g. L2/L3) are typically slower than those closer to the CPU, but also much larger [12]. The first few levels of caches are often local to each CPU core; the last cache level, aptly called the last-level cache (LLC), is often instead shared across all cores on the same CPU socket. An abusive use of the LLC by

 DOI: 10.1109/CCGrid54584.2022.00059

an application running on one CPU core can lead to cache contention. In FaaS, for density purpopes, multiple functions are executed on the same server, thus sharing the same LLC. Thus, if a function is abusively using the LLC, its activity can lengthen the execution of other functions due to cache contention. To address cache contention in the LLC, Intel introduced a new technology known as Cache Allocation Technology [13], [14] (*CAT* for short), which provides software-programmable control over the amount of cache that can be used by a process, virtual machine or container. However, no FaaS platform uses this CAT technology to allocate CPU cache to functions. In this work, we propose a system named *CASY* for ***C**PU cache **A**llocation **SY**stem*, that leverages CAT to allocate the CPU cache to functions in a FaaS platform.

Our *CASY* solution allocates the CPU Cache to functions according to their needs. The building of such a system raises two challenges: *Challenge #1*. How to determine the amount of CPU cache needed by a function? Indeed, the LLC usage of a function depends not only on its implementation, but also on the input data. For example, a function will likely not have the same cache requirement when processing a 100 KBytes file versus a 200 MBytes file. This element makes cache allocation more complex. *Challenge #2*. How to allocate the LLC to functions knowing that it is a limited resource? It is very likely that the size of the LLC is not sufficient to satisfy the LLC requirements of all running functions. Consequently, the following question stands out: how to allocate the LLC to these functions in other to get the fastest execution possible for all functions? To answer these two challenges, *CASY* integrates two components: (1) *CASY ML*, which is a machine learning module that aims to build models that can predict the LLC requirement of a function according to the input data, and (2) a *CASY cache allocator* which is the component that allocates LLC to functions according to their predicted cache requirement and the current LLC load level.

CASY Machile Learning (ML). We leverage machine learning to determine the amount of LLC needed by a function according to the input data. We define two types of function with respect to the cache usage: Cache Sensitive functions (*CS* for short) and Cache Insensitive functions (*CI* for short). *CS* functions are impacted by CPU cache contention, whereas *CI* functions undergo little or no impact on performance due to cache contention. Note that a function can be both CS or CI depending on the input data. Thus, for a given function with a given input data file, the ML module allows to determine if it is CS or CI and also to compute the amount of LLC needed for the function. To do so, *CASY ML* builds ML models for each function. We noticed from our study and also based on previous work [7] that the memory and cache activity of a function depends on the size of the input data. Thus, for a function, its models was build using the size of the data as input variable.

CASY Cache Allocator. The cache allocator is the module that allocates the LLC to each function. After *CASY ML* has computed the amount of LLC to allocate, the allocation module allocates the LLC to the functions depending on their needs and also the system load. Globally, the cache is divided into ways and these ways can be shared among the cores with CAT. The *CASY cache allocator* satisfies functions by allocating the required number of ways but also ensure that the global load is equally distributed on the cache ways. To do so, the *CASY* cache allocator assigns a load to each cache way which represents its occupation by functions. Each time a new function is scheduled, the allocator allocates the least used cache ways to this function, allowing the load to be distributed equally over all cache ways.

We implemented our system and integrated it in the OpenWhisk FaaS platform, and we thoroughly evaluated it with various functions, such as functions that perform image and video processing. In summary, the contribution of this paper are as follows:

- we present *CASY*, a CPU cache allocation system targeting FaaS platforms which uses Intel CAT to allocate the LLC to functions depending on the input data size;
- *CASY* uses ML models to predict the amount of CPU cache needed by each function;
- *CASY* implements an allocation strategy that distributes the load on LLC ways and guarantees good performances to functions;
- we evaluated *CASY* with several functions, and we observed a performance improvement of up to 11% for some functions without degrading the performance of other functions.

The rest of our article is organized as follows. Section II presents the necessary background to understand our contribution. Section III presents our observations and motivations for the creation of *CASY*. Sections IV to V present the design and implementation of *CASY*. Section VI presents the evaluation results. Section VII discusses the related works. Section VIII concludes our paper.

II. Background

In this section, we introduce all the necessary background to understand our solution.

A. FaaS: Function-as-a-Service

Serverless computing is increasingly becoming a must-have service for all cloud providers, and is turning into a popular approach to deploy applications in the cloud. The setup of a FaaS can be done on a private clouds with a server cluster using frameworks such as OpenWhisk [6], OpenFaas [15], etc. In this relatively new cloud service, users provide functions written in languages such as Python, Javascript, Go, Java etc., and these functions are executed by the FaaS platform. This new paradigm greatly simplifies resource management for applications.

In FaaS, the state of functions is not persistent across invocations. Therefore, function definitions start by importing and loading all code and data dependencies using the ETL model (see Section I). Each function is executed in a sandbox, which can be a container such as Docker [3], or a lightweight VM such as Firecracker [4]. By encapsulating all of the

function state and any side-effects, the sandbox environment provides isolation among multiple functions, and also allows for concurrent invocations of the same function. The FaaS platform is responsible for resource allocation to the sandbox. Today, in most FaaS platforms such as OpenWhisk, the scheduler is responsible for assigning each function to a server for execution based on the CPU and memory available and also from the availability of hot (pre-started) containers that can quickly run the function. None of these platforms integrate CPU cache allocation, even though this component can impact function execution time as we have shown in Section III.

B. CAT: Cache Allocation Technology

The CAT technology is part of Intel's Resource Director Technology (RDT). It provides a way to allocate a given amount of CPU cache to a process, virtual machine or container. A operating system or hypervisor can use CAT to protect important applications and virtual machines from noises originating from cache usage fluctuations, and also to implement resource prioritization. How does CAT work?

Intel CAT is used to allocate ways of the LLC. Therefore, the allocation unit of CAT is a cache way. Each cache way can be allocated to several processes. To carry out the allocation, CAT introduces an intermediate structure called Class Of Service (CLOS), which acts as a resource control plane in which a number of ways can be associated and assigned to a process. Each CLOS corresponds to a capacity mask (CBMs), in which a *1* bit means that a process belonging to this CLOS can allocate the specified cache way, and a *0* bit means the opposite. This therefore allows to control the amount of cache that a process in a CLOS can allocate. Several process can be associated to the same CLOS. Depending on the CPU architecture, it is only possible to define a limited number of CLOS. For the hardware use in this paper, we were limited to 8 CLOS and 11 ways (See Section VI for details).

III. Motivation

To motivate the need for CPU cache allocation in FaaS environments, we performed two evaluations: one on the impact of cache contention on function execution time and one on the impact of Intel CAT. The experimental environment is the same as described in Section VI; the types of functions used are also described in this section.

We executed 4 functions (Effect, Blur, Speech Recognition, Resize) on a server in two scenarios: firstly, each function is executed alone on the server; then the 4 functions are executed simultaneously on the same NUMA node. To ensure no system overhead, the execution environments of the functions are pinned to individual CPUs. The obtained results are highlighted in Fig 1. We can notice that the execution time of two functions: Blur and Speech Recognition decrease by 37 % and 16% respectively, while others functions are less impacted. As we will see later, this performance variation is caused by cache contention from the two other functions Effect and Resize. This motivates the need for a system that allows to isolate the functions from each other.

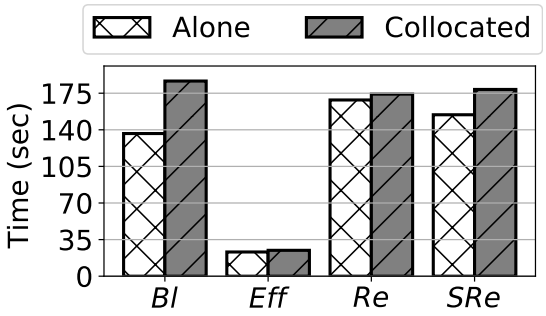

Fig. 1. Execution time of 4 FaaS functions running alone and collocated.

We also executed our functions individually with different data sizes while using Intel CAT to allocate various amounts of cache. The results obtained are presented in Fig 2. We can observe that for Blur, when the input size is 1.4 MBytes, the function is impacted by the amount of cache allocated; but when the input size is reduced, the function becomes less sensitive to cache allocation. Regarding Speech Recognition, this function stays sensitive to the cache regardless of the data size. Conversely, Effect and Resize are insensitive to cache allocation regardless of data size. The observations we can make from these evaluations are:

- cache contention can impact the execution time of functions;
- the use of the cache by functions depends on the input data;
- a function can be cache sensitive with a particular input data and insensitive with another. This behavior makes the allocation of CPU cache resources more complex.

It therefore appears interesting to propose a system that allows to allocate the CPU cache to functions, and this system should be easily integrated to FaaS platforms.

IV. Contribution

As an answer to the CPU cache allocation problem, we *CASY*, a system which leverages machine learning to profile the CPU cache behavior of functions and subsequently uses the ML model to predict and allocate the CPU cache by using the Intel CAT technology. Fig 3 depicts the overall architecture of our system and the interaction flow with a FaaS platform.

A. Overview

Fig 3 shows the general architecture of *CASY*, which consists of two main components (green boxes in Fig 3): *CASY* ML and *CASY* Cache Allocator. The ML component performs the machine learning part, i.e. learning and predicting the amount of LLC cache needed for a function with a given input data. To this end, it consists of two features: the *profiler* and the *predictor*. The purpose of the profiler is to build models for each function. Whenever a new function is stored in the registry/data base (step ❶), the profiler fetches the functions and executes it with various data sizes (step ❷) on a dedicated server. These various runs allow building ML models for the function; our models use the data size as an input variable.

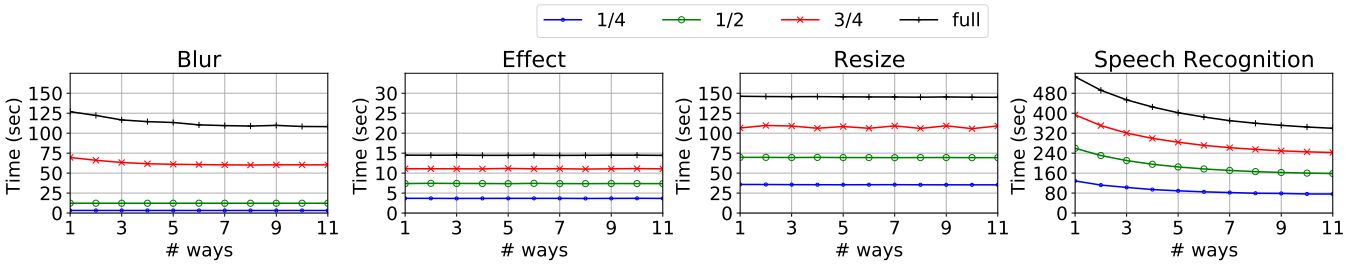

Fig. 2. Impact of Cache allocation with CAT on function execution time with various input file sizes. 1/4, 1/2, 3/4 represents the file sizes as ratios of the initial size which are 1.4 MBytes for Blur, 100 MBytes for Effect, 127 MBytes for Resize and 79 MBytes for Speech Recognition.

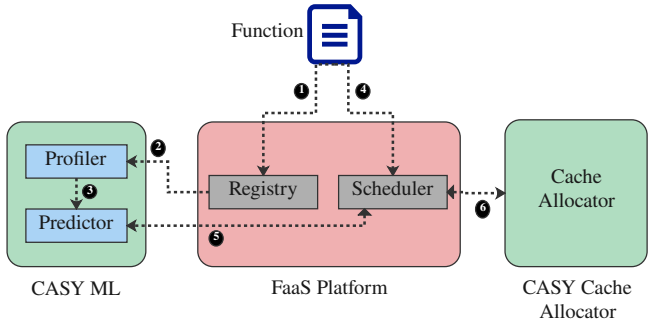

Fig. 3. General architecture of *CASY*, including its components and its interaction with a FaaS platform.

This operation is performed offline, i.e. without the occurrence of the events which trigger the function. Once the models are built, they are forwarded to the predictor (step ❸) for future allocation. Each time a function is invoked (step ❹), the FaaS platform scheduler addresses the predictor to provide the required cache size (step ❺). Once the prediction is made, the scheduler turns to the *CASY* cache allocator to configure the execution environment (sandbox) of the function (step ❻). The cache allocation is performed taking into account the requirements of the function as well as the actual cache load. In the rest of the sections, we detail the functioning of each component of *CASY*.

B. CASY *ML*

Before presenting the ML module, it is necessary to recall our classification of functions with respect to the CPU cache usage. As we stated in Section I, we defined two classes of functions, *CI* and *CS*:

- *CS* for functions sensitive to cache contention. These are functions for which cache contention has a substantial impact on their execution time. Their data fits in the LLC cache and they perform very few cache misses.
- *CI* for functions insensitive to cache contention. In this class, we distinguish two types of functions: those whose data do not fit in the cache and thus pollute neighboring workloads (cache-thrashing functions), and those that use little cache and are thus not affected by cache contention.

It is important to note that a function can be *CI* or *CS* depending on the input data. The class of a function is defined accordilying with an input data. These two classes will be useful for allocating cache ways.

The goal of the profiler is twofold: (1) to build a model that can predict the number of cache ways required by a function with a given input data, and (2) to build a model that can specify whether a function is *CI* or *CS*. Regarding the first goal, the profiler builds a ML model from prior executions of the function with diverse datasets on a dedicated server. We call this *function profiling*. The function profiling is performed once and should be carried out on a free server to avoid cache contention. We assume that generally a function is invoked frequently and over a long time period. Hence, the cost of profiling is minimal compared to the long term gain. In *CASY*, we assume that whenever a function is registered, the client may either provide a test dataset on which we can train and build the model, or provide the type of input data in order for *CASY* to generate the needed data. We observed from our results and from previous work [7] that the input file size is a sufficient property to capture the CPU cache behavior of a function. The function profiling is performed following Listing 1. It involves executing the function with various file sizes (lines 12 and 13 of Listing 1) while reducing the number of allocated cache ways (lines 11 and 12). The aim is to identify, for each file size, the minimal number of ways that allows obtaining the same level of performance as when the function has full access to the cache (line 5). To this end, we define a threshold (5% in our article) which represents the level from which we consider that there is an impact on the execution time. For a function with a given input file, we progressively reduce the number of allocated cache ways and define the lowest number of cache ways possible while staying below the performance threshold. In line 14 we compute the degradation (named Δ in the listing) that we compare with our threshold in line 10

Once this execution step is done, we then have for each function a pair of values $(size, nr)$, where $size$ is the input file size and nr is the associated minimal number of cache ways. Subsequently, we use this value pairs to construct an ML model. Our ML model uses the file size as an input variable and returns the number of cache ways.

The second goal of the profiler is to build a model that can predict the class of a function based on the input data size. In order to do this, the profiler uses the cache sensitivity of a

Algorithm 1 Function profiling algorithm

```
func ← GetFunction( )
L_files ← GetInputFileList( )
N_ways ← GetMaxWays( )
for file in L_files do
    Allocate(N_ways)
    Δ ← 0
    T_Ref ← GetExecTime(func, file)
    size ← Sizeof(file)
    nb ← N_ways + 1
    while Δ < Threshold and nb > 1 do
        nb ← nb - 1
        Allocate(nb)
        T_act ← GetExecTime(func, file)
        Δ ← (T_Ref − T_act) / T_Ref × 100
    end while
    SetNbWays(size, nb + 1)
end for
```

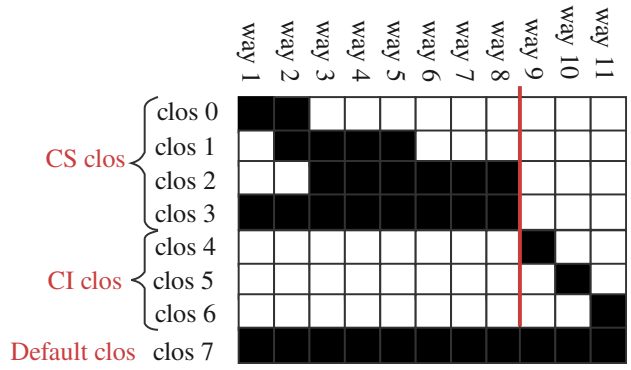

Fig. 4. An example of a 8-CLOS configuration with a 11-way CPU.

function. We define the sensitivity S of a function f with a particular input data d as the performance degradation when the function has full access to the entire LLC compared to when it has access to only one way of the cache. Equation 1 presents the formula for S with $T_{f(d)}^{N_{ways}}$ and $T_{f(d)}^{1}$ being the execution time with the full LLC and a single cache way respectively.

$$S_{f(d)} = \frac{T_{f(d)}^{N_{\max}} - T_{f(d)}^{1}}{T_{f(d)}^{N_{\max}}} \times 100 \tag{1}$$

A function is said to be *CI* if its sensitivity is above a threshold (we consider 5% to be a fairly significant degradation). For each function, the results from different runs with various data are used to build an ML model which will take as entry the size of the data and classify them as *CI* or *CS*.

C. CASY *Cache allocator*

As we mentioned in Section II, the CPU architecture has a fixed number of CLOS that can be set and number of LLC ways that can be assigned. Consequently, it is important to properly set up these CLOSes. On all servers, we statically setup the CLOS configuration and associated cache ways. We split our LLC into two chunks: one for the *CI* functions and another for the *CS* functions. Let's call N_{ways} the total number of cache ways available for a CPU and N_{CLOS} the number of CLOS that can be set. We split N_{ways} into two chunks: N_{CI} and N_{CS} where $N_{ways} = N_{CI} + N_{CS}$. This setup aims at keeping the *CI* functions and the *CS* functions separate in order to reduce the cache contention. The rest of the configuration deals with assigning ways to CLOS.

- For the N_{CI} ways, they are set to single-way CLOS. Since these functions are not impacted by the number of allocated cache ways, their execution environments will be configured with a single way.
- For the N_{CS} ways, we configure the associated CLOS following two rules: (1) the cache ways will be allocated to the smallest possible number of CLOS and (2) the CLOS will span as many as possible combinations with the available ways.
- Finally, a *default CLOS*, which contains all cache ways, will be defined for functions that do not have ML models.

To illustrate this, let's consider an 11-way LLC for a CPU architecture on which we can configure 8 CLOSes as shown in Fig 4. Thus, if we decide to set three ways for *CI* functions and eight for *CS* functions, we obtain the configuration shown in Fig 4. Each CLOS for *CI* functions has a single way (see Fig 4). As for the CLOS for *CS* functions, we have eight remaining available cache ways. We configure the CLOS from two to eight ways, where one way is used in a maximum of three CLOSes. Finally, the *CLOS 7* (the default CLOS) which includes all cache ways will be used for functions not having ML models.

The allocation of CLOSes to functions is done based on the load on the CLOS and also the predicted number of ways required for the function. Before presenting the allocation heuristic, it is important to define three metrics: the weight of a function, the load on a cache way and the load on a CLOS. We define the weight of a function f with data d with respect to a CLOS of N_{COS} ways in Equation 2. $P_{f(d)}$ is the number of ways that is provided by the predictor.

$$W_{f(d)} = \frac{P_{f(d)}}{N_{\text{COS}}} \tag{2}$$

The load on a cache way is defined as the sum of all the function weights that use this way. Equation 3 presents the formula for calculating the load on a way.

$$L_{way} = \sum_{n=1}^{nb_{\text{Func}}} W_{f_i(d)} \tag{3}$$

The load of a CLOS is defined in Equation 4 and is the sum of the loads of cache ways assigned to the CLOS.

$$L_{\text{COS}} = \sum_{n=1}^{nb_{\text{Way}}} L_{way_i} \tag{4}$$

Listing 2 presents our allocation heuristic. Each time a function is invoked, the scheduler queries the predictor, indicating the function and the data size. Two cases are then possible:

- the models associated with the function are not available, so the execution environment is simply set to the default CLOS (line 5);
- the models exist and the predictor provides the number of ways needed and the function class (line 3). This is the interesting case that we detail in the following paragraph.

If the function is a *CI*, then its execution environment will be configured with a CLOS that is dedicated to these functions (line 9); otherwise, we use the CLOSes that are reserved for *CS* (lines 11). We then compute the potential new load on all cache ways with the new function weights (line 14), and sort the list of CLOSes according to this new potential loads (line 16).We then ensure that if there are several CLOSes with a load less than 1, then among these CLOSes the one with the highest load is chosen, to avoid choosing a CLOS with more ways than necessary (line 18 to 21). If not, the selected CLOS will be the one with the lowest load (line 23). At the completion of the function execution, the loads of the CLOS and ways are updated.

Algorithm 2 Get the appropriate CLOS for a function

```
1: func ← GETFUNCTION( )
2: size ← GETINPUTFILESIZE( )
3: P_f, CL ← PREDICTNBWAYS(func , size)
4: if P_f == null then
5:     S_CLOS = GETDEFAULTCLOS( )
6:     return S_CLOS
7: end if
8: if CL == "CI" then
9:     list_clos ← GETCLOSLIST("CI")
10: else
11:     list_clos ← GETCLOSLIST("CS")
12: end if
13: list_ways ← GETALLWAYS(list_clos)
14: list_ways ← ADDLOADWAYS(list_ways , func , P_f)
15: list_clos ← UPDATEALLCLOSLOAD(list_clos , list_ways)
16: list_clos ← SORTBYLOAD(list_clos)
17: for clos in list_clos do                ▷ list_CLOS is sorted
18:     if GETLOAD(clos) ≤ 1 then
19:         S_CLOS ← clos
20:     end if
21: end for
22: if S_CLOS== "Null" then
23:     S_CLOS ← list_clos[0]
24: end if
```

V. IMPLEMENTATION OF *CASY*

The main challenge of *CASY* implementation is to offer a system that can be easily integrated with FaaS platforms. That's why we designed *CASY* as a standalone system that can interface with FaaS platforms. *CASY* is fully implemented in Python. The rest of this section will focus on the implementation of the two components of *CASY* and the integration in Apache OpenWhisk.

A. CASY *ML*

This component consists of two features: the profiler and the predictor. The profiler extracts functions from the registry or database (Apache CouchDB in the case of Open Whisk) and executes them with various input data. The profiling is done by executing functions on a dedicated server, all this offline. We suppose that a server will be dedicated to function profiling when needed, or a NUMA node on the server. The profiling data can be provided by the user or by *CASY*. *CASY* currently has videos and images that can be used to train functions. Regarding the ML, we used *k-NN clustering as model* with scikit-learn library. Given that for the two models to be built per function, it involved prediction towards discrete universes: from 1 to the maximum number of ways and the selection between *CI* and *CS*, we selected a supervised learning. It should be noted that with the ML model (as with almost all ML models) the more diverse the training data set is, the more accurate the model will be. The evaluation section shows that in our case, a relatively small set of training data allows us to have a good level of accuracy with K-NN.

B. CASY *Cache Allocator*

This component is responsible for the configuration of the execution environment. Currently, the implementation that we have uses Intel pqos tool [16] to configure CLOS and rdtset [16] to associate a CPU to a CLOS. This library allows defining CLOS, ways associated with them, and also to link CPUs to CLOS. Thus, on all servers, we use these libraries to set up all CLOS and afterward, at function invocation, *CASY* cache allocator add the selected CPUs to execute a function to the CLOS designated by his algorithm. The current version of *CASY* allows configuring CLOS for containers.

C. Integration into Apache OpenWhisk

Apache OpenWhisk is a very popular open source FaaS manager. We describe the integration of *CASY* on this FaaS management. Let's start by introducing the main components of OpenWhisk:

- a Nginx server which is the entry point for requests/events;
- a controller that translates requests into function invocations, and includes a load balancer that chooses on which server the function will be executed. The load balancer also defines the configuration parameters of the execution environment, i.e CPU pinning and amount of memory;
- a Kafka server that serves as a communication tool between the controller and the invokers on servers;
- a CoucheDB (the registry) that contains functions;

The integration of *CASY* in such a system is quite simple, as the *CASY* ML interacts with CoucheDB to extract the functions, execute them, and save the training data. At the Controller level, since it is the Load Balancer that decides and configures the execution environment, we hacked the invocation communication to know which functions are to be executed and the input data size. Thus we can use the predictor to determine the class and the number of required ways. Then,

based on the settings provided by the Load Balancer, i.e the choice of the CPUs where the container is going to be pinned, *CASY* allocator configures the affinity of this CPU with the CLOS designated by its allocation algorithm.

VI. Evaluation

In this section, we present the performance evaluation of *CASY*. With these evaluations, we plan to answer the following questions:

- what is the precision of *CASY* ML module?
- what is the improvement brought by *CASY* on the execution time?
- what is the Overhead of *CASY*?

Therefore, to evaluate *CASY* we proceeded in two main steps: The validation of our ML model and the evaluation of our allocation system.

A. Experimental setup

a) Hardware: We evaluated *CASY* on a server with the characteristics described in Table I. The CPU of the server has CAT and allows the configuration of 8 CLOS. We have configured the CLOS as in Section IV.

Component		Characteristics
CPU	Cores	Intel(R) Xeon(R) Silver 4210R CPU
	Level 1 cache	20×32KB, 8-ways
	Level 2 cache	20×1MB, 16-ways
	Level 3 cache	Shared, 2×13.75MB, 11-ways
Memory		64 GBytes
Ethernet		Intel X710 10GbE
Storage		1.0 TB HDD
OS		Ubuntu 20.04.3, Linux kernel 5.4.0

TABLE I
HARDWARE CONFIGURATIONS.

b) Applications: We evaluated *CASY* using several functions. Table II provides the list of functions used in our experiments and their descriptions. We used the same deployed functions as in papers dealing with performance in FaaS [7].

B. Machine Learning model validation

This step consists in verifying our prediction model accuracy. For each function specified above (in Table II), we performed training with a set of files, 4 files with characteristics listed in Table III, and then evaluate the prediction model with 10 different files. Table III presents the size of the files used for each function as well as the number of ways detected as required by our profiling algorithm for these various sizes. We can observe that Effect and Resize functions always require 1 cache way. They are *CI* functions while Blur and Speech Recognition have their cache demand varying with the input file size. More particularly, Blur sees its requirement in the number of ways increase from 1 to 7 while passing by 4.

Function	Description
Blur	Takes as input a PNG image file and applies a blur
Effect	Takes as input an audio file in WAV format and applies a sound effect
Resize	Takes a high resolution video and return a lower resolution video
Speech Recognition	Takes an audio file and performs voice recognitions

TABLE II
FUNCTION DESCRIPTIONS.

Once the models are built the next step is to validate them. To validate our models, we used the model to predict the number of ways required with different file sizes and compared the results to a run with full access to the LLC. Figure 5 presents the obtained results. Each bar plot represents a file size used and the execution time obtained when the function has full access to the cache and when it has access to the amount predicted as needed. The number above each bar represents the number of ways predicted by the models. We can observe that for all the functions, the execution times with *CASY* and with all the cache are identical. This means that the number of cache ways predicted as required by *CASY* seems to agree with the real needs for the applications. And also, for all functions, the model has always been able to determine the class of a function, that is; Blur and Speech Recognition are *CS* and Effect and Resize are CI. We can observe a non-proportionality between some file sizes and the execution time for blur and speech recognition loads. This can be explained simply by acknowledging that some large files have a lower resolution and thus induced a shorter processing time.

Effect		Blur	
Size	# Ways	Size	# Ways
26 MB	1	124 KB	1
51 MB	1	361 KB	1
76 MB	1	699 KB	4
100 MB	1	1.4 MB	7
Speech Recognition		**Resize**	
Size	# Ways	Size	# Ways
40 MB	10	31 MB	1
79 MB	11	61 MB	1
118 MB	11	92 MB	1
157 MB	11	127 MB	1

TABLE III
REQUIERED AMOUNT OF CACHE FOR EACH TRAINING FILE SIZE FOR OUR BENCHMARKS

C. Allocation evaluation

For the evaluation of our allocation algorithm, it was necessary to replay execution scenarios of functions from a datacenter. As we did not have such data, we implemented an emulator that follows a probability distribution, the Poisson distribution, to generate execution scenarios. In other words, we used the Poisson distribution to generate the start time of functions, and randomly choose the function to execute and the file to use. The Table IV shows the various input data sizes used with each function. We also present in the table the acronym that we use on our figures. Thus, we used our generator to generate scenarios involving 6, 10 and 20 functions running simultaneously with and without *CASY*. Each scenario is executed 10 times and the presented

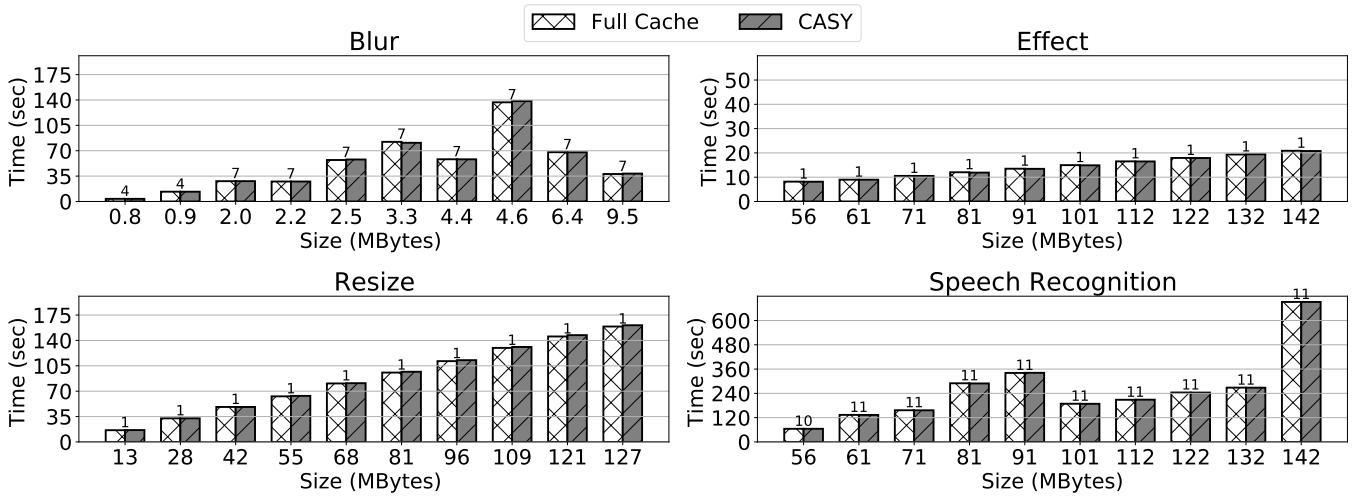

Fig. 5. ML model evaluations with all functions using various input file sizes

results are the average execution time of functions. Fig 6 shows the results for some of these scenarios. Let's start our analysis with the execution involving 6 functions (first row of the figure). We can notice that Blur is the one that shows an improvement with our solution. This improvement is about 11% (red circles in the figure). Moreover, we can also observe that Resize, which is executed with Blur, is not impacted with *CASY*. This shows that *CASY* has been able to identify that Resize is a *CI* function and to make an allocation accordingly. Let's continue our analysis with the execution with 10 functions (second row on the figures). We can notice that the function that has the most impact is still Blur (7%) followed by Speech Recognition (5%). As we noted in the previous section, Speech Recognition requires 11 cache ways which is the total number available on our server. Thus, the impact of our allocation system remains limited on this benchmark. Finally, the last scenario is about the execution of 20 functions simultaneously (third row on the figure). We can see that there are little improvements on the performance for Blur and Speech Recognition. These improvements are less than in the scenarios with fewer functions (6 and 10). This is because the cache is a limited resource and the more they are functions using the cache, the less the impact on allocating the cache is perceptible. Nevertheless, we notice that *CASY* does not lead to any performance degradation for other functions.

Acronym	Configuration
Bl_1	Blur with file size 4.6 MB
Bl_2	Blur with file size 6.4 MB
Bl_3	Blur with file size 9.5 MB
Re_1	Resize with file size 37 MBytes
Re_2	Resize with file size 67 MBytes
Re_3	Resize with file size 127 MBytes
SRe_1	Speech Recognition with file size 79 MBytes
Eff_1	Effect with file size 100 MBytes

TABLE IV
FUNCTION WITH THE ASSOCIATED FILE SIZES AND ACRONYM

D. *Overhead of* CASY

The overhead of *CASY* comes mainly from the profiling, as prediction and configuration of CLOS consume almost no resources (CPU and memory). The function profiling process is the most important overhead of *CASY* because it requires running the function on a dedicated server (or on a dedicated NUMA node on a server) with different file sizes. As aforementioned, we consider that the lifetime of a function is quite long and a function is most likely invoked several times. Thus, we consider that this time needed for profiling is compensated with the benefits that come from the CPU cache allocation.

VII. RELATED WORK

The studies related to our work were mainly carried out in two areas: optimization of the last level cache management and improvement of the start-up and execution time of functions in FaaS environments.

a) Last Level Cache management: Intel's Cache Allocation technology has opened the door to many possibilities for optimizing processor cache management. The vast majority of recent research targeting the improvement of processor cache usage relies on this feature. [14] showed the benefits of using CAT to protect Virtual Network Function (VNF) resources against "Noisy Neighbor" effects, by deterministically prioritizing LLC resources between competing workloads. [17] proposes dCAT, a dynamic cache management technology to provide cache isolation with better performance. [18] proposes CP_{pf}, a prefetch aware LLC partitioning approach for improving LLC management. [19] devises a cache allocation scheme from an empirical analysis of different operators and integrate a cache partition mechanism in a commercial database management system. [20] introduces slice-aware memory management scheme and proposes CacheDirector, a network I/O solution that extends Direct Data I/O and places the packet's header in the slice of the LLC that is closest to the relevant processing core. [13] proposes an approach that automatically builds a prediction model for application

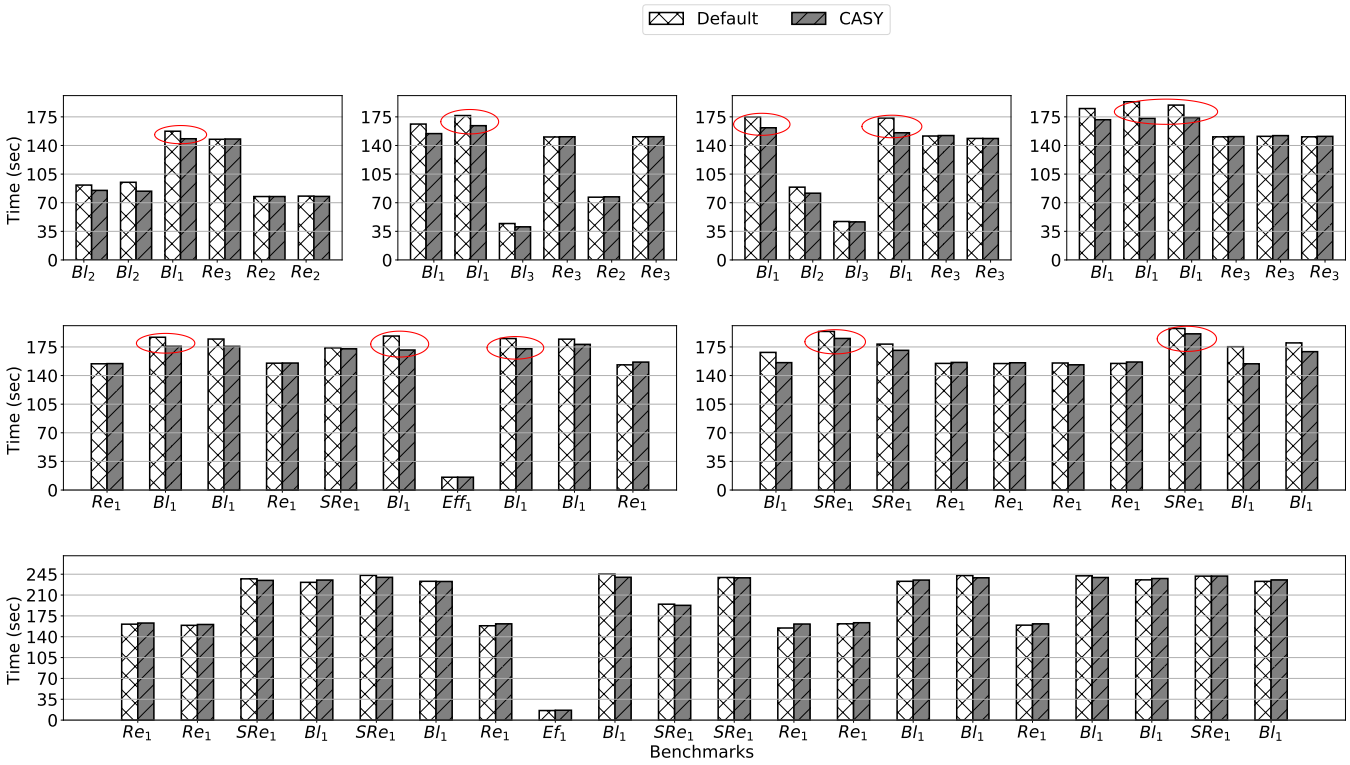

Fig. 6. Allocation evaluation

performance changes with CAT, and a dynamic cache management technique that utilizes that prediction model and intelligently partitions the cache resource to improve application throughput. DCAPS [21] dynamically monitors and predicts a multi programmed workload's cache demand to reallocate LLC given a performance target, and explores partial sharing of a cache partition among programs to achieve cache allocation at a finer granularity. A family of clustering-based cache partitioning policies was proposed by [22] to address fairness using CAT.

b) Reducing the start-up latency and execution time: To reduce the start-up latency and the execution time of a function, some works are interested in the start-up time optimization of the execution environment, while others are focussing on the scheduling of functions. [23] proposes a greedy load balancing algorithm optimized for FaaS which provides higher locality. OpenWhisk [24] also uses an optimized load balancing algorithm that considers locality and container reuse to minimize both startup latency (cold start) and execution time. [25] implemented three approaches that reduce the number of cold starts while treating the FaaS service as a black box. In these approaches, they use the knowledge on the composition of functions to trigger the provisioning of new containers before the application process invokes the respective function. OFC [7] is an elastic in-memory caching system for FaaS platforms, that estimates the actual memory resources required by each function invocation and hoards the remaining capacity to feed the cache, using machine learning models adjusted for typical function input data categories. [26] proposes a package-aware scheduling algorithm that tries to assign functions that require the same package to the same worker node, thereby increasing the hit rate of the package cache and, consequently, reducing the start-up latency of cloud functions. [27] proposes a function startup technique, which restores snapshots of previously executed function processes. FaaSRank [28] is a function scheduler for serverless FaaS platforms based on information monitored from servers and functions, which automatically learns scheduling policies through experience using reinforcement learning (RL) and neural networks. [29] proposes guidelines for orchestrating massively parallel workloads using serverless functions to reduce overheads. FnSched [30] is a function-level scheduler designed to minimize provider resource costs while meeting customer performance requirements, which works by carefully regulating the resource usage of colocated functions on each invoker, and autoscaling capacity by concentrating load on few invokers in response to varying traffic. [31] proposes a new cluster-level centralized and core-granular scheduler for serverless functions which improves elasticity and reduces interference.

c) Position of our work: The innovation of our work compared to the existing one is that we combine LLC management with machine learning in a FaaS context. We take advantage of the fact that FaaS functions are lightweight environments with coherent cache usage, to classify them according to their cache usage profile, and thus perform smart

cache allocation.

VIII. Conclusion

In this article, we introduced *CASY*, a solution for enhancing the CPU cache allocation to functions in a FaaS platform. *CASY* leverages ML to build CPU cache usage profiles for functions according to the input data, and uses CAT to partition cache. The function profiling is done by executing functions with various data and on a dedicated server. Once the profiling is done, *CASY* uses a classifier to build a model that can predict the cache requirement of a function according to its input data. *CASY* also proposes an algorithm for cache allocation which allows a balanced spread of the loads on all the cache ways. *CASY* is built as a system that can easily be integrated with FaaS platforms. We evaluated the performance improvement induced by *CASY* under multiple different scenarios and showed that *CASY* can reduce the execution time of some functions while maintaining the same execution time for other functions.

Acknowledgments

This work is supported by the French *Agence nationale de la recherche* under the ANR WalkIn (ANR-20-CE25-0005) project.

References

[1] B. Hayes, "Cloud computing," 2008.

[2] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 162–169.

[3] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[4] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, 2020, pp. 419–434.

[5] D. Kelly, F. G. Glavin, and E. Barrett, "Serverless computing: Behind the scenes of major platforms," *CoRR*, vol. abs/2012.05600, 2020. [Online]. Available: https://arxiv.org/abs/2012.05600

[6] M. Sciabarrà, *Learning Apache OpenWhisk: Developing Open Serverless Solutions*. O'Reilly Media, 2019. [Online]. Available: https://books.google.fr/books?id=gEqgDwAAQBAJ

[7] 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*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 228–244. [Online]. Available: https://doi.org/10.1145/3447786.3456239

[8] A. AWS, "Amazon lambda pricing," https://aws.amazon.com/lambda/pricing/, 2021.

[9] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 386–400. [Online]. Available: https://doi.org/10.1145/3445814.3446757

[10] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 218–233. [Online]. Available: https://doi.org/10.1145/3132747.3132763

[11] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, jul 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407836

[12] A. J. Smith, *Design of CPU cache memories*. Computer Science Division, University of California, 1987.

[13] Y. Kim, A. More, E. Shriver, and T. Rosing, "Application performance prediction and optimization under cache allocation technology," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1285–1288.

[14] P. Veitch, E. Curley, and T. Kantecki, "Performance evaluation of cache allocation technology for nfv noisy neighbor mitigation," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, 2017, pp. 1–5.

[15] C. Kaewkasi, *Docker for Serverless Applications: Containerize and orchestrate functions using OpenFaas, OpenWhisk, and Fn*. Packt Publishing Ltd, 2018.

[16] "Intel(r) rdt software package," https://github.com/intel/intel-cmt-cat/, last Accessed: Dec 10, 2021.

[17] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dcat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," 04 2018, pp. 1–13.

[18] J. Xiao, A. Pimentel, and X. Liu, "Cp$_{pf}$: a prefetch aware llc partitioning approach," 08 2019, pp. 1–10.

[19] S. Noll, J. Teubner, N. May, and A. Böhm, "Accelerating concurrent workloads with cpu cache partitioning," 04 2018, pp. 437–448.

[20] A. Farshin, A. Roozbeh, G. Jr, and D. Kostic, "Make the most out of last level cache in intel processors," 03 2019.

[21] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: dynamic cache allocation with partial sharing," 04 2018, pp. 1–15.

[22] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," 09 2017, pp. 194–205.

[23] Y. Lee and S. Choi, "A greedy load balancing algorithm for faas platforms," in *2021 5th International Conference on Cloud and Big Data Computing (ICCBDC)*, ser. ICCBDC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 63–69. [Online]. Available: https://doi.org/10.1145/3481646.3481657

[24] "Apache openwhisk is an open source, distributed serverless platform," https://openwhisk.apache.org/, last Accessed: Dec 10, 2021.

[25] 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*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 134–143. [Online]. Available: https://doi.org/10.1145/3341105.3373909

[26] C. L. Abad, E. F. Boza, and E. van Eyk, "Package-aware scheduling of faas functions," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 101–106. [Online]. Available: https://doi.org/10.1145/3185768.3186294

[27] 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. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3423211.3425682

[28] H. Yu, "Faasrank: A reinforcement learning scheduler for serverless function-as-a-service platforms," Ph.D. dissertation, University of Washington, 2021.

[29] D. Barcelona-Pons, P. García-López, A. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, "Faas orchestration of parallel workloads," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–30. [Online]. Available: https://doi.org/10.1145/3366623.3368137

[30] A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 19–24. [Online]. Available: https://doi.org/10.1145/3366623.3368136

[31] 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. New York, NY, USA: Association for Computing Machinery, 2019, p. 158–164. [Online]. Available: https://doi.org/10.1145/3357223.3362709