OFC: an opportunistic caching system for FaaS platforms

Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ndjie, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al.

OFC: An Opportunistic Caching System for FaaS Platforms

Djob Mvondo
Univ. Grenoble Alpes
ENS Lyon

Mathieu Bacou
Telecom SudParis

Kevin Nguetchouang
Lucien Ngale
ENSP Yaoundé

Stéphane Pouget
ENS Lyon

Josiane Kouam
Inria

Renaud Lachaize
Univ. Grenoble Alpes

Jinho Hwang*
Facebook

Tim Wood
The George Washington University

Daniel Hagimont
University of Toulouse

Noël De Palma
Univ. Grenoble Alpes

Bernabé Batchakui
ENSP Yaoundé

Alain Tchana
ENS Lyon
Inria

Abstract

Cloud applications based on the "Functions as a Service" (FaaS) paradigm have become very popular. Yet, due to their stateless nature, they must frequently interact with an external data store, which limits their performance. To mitigate this issue, we introduce OFC, a transparent, vertically and horizontally elastic in-memory caching system for FaaS platforms, distributed over the worker nodes. OFC provides these benefits cost-effectively by exploiting two common sources of resource waste: (i) most cloud tenants overprovision the memory resources reserved for their functions because their footprint is non-trivially input-dependent and (ii) FaaS providers keep function sandboxes alive for several minutes to avoid cold starts. Using machine learning models adjusted for typical function input data categories (e.g., multimedia formats), OFC estimates the actual memory resources required by each function invocation and hoards the remaining capacity to feed the cache. We build our OFC prototype based on enhancements to the OpenWhisk FaaS platform, the Swift persistent object store, and the RAMCloud in-memory store. Using a diverse set of workloads, we show that OFC improves by up to 82 % and 60 % respectively the execution time of single-stage and pipelined functions.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Software and its engineering → n-tier architectures**.

*A part of work done while Jinho Hwang was at IBM Research.

EuroSys '21, April 26–28, 2021, Online, United Kingdom

Keywords: cloud computing, serverless, functions as a service (FaaS), cache, latency

ACM Reference Format:
Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3447786.3456239

1 Introduction

Over the past few years, the *Function as a Service* (FaaS) paradigm has deeply reshaped the way to design, operate and bill cloud-native applications. It allows tenants to focus only on the application code, which is organized as a set of stateless functions. Furthermore, resources are charged to the user only when they are actually used. The user just uploads the functions, which will be automatically triggered by events (e.g., timer, HTTP request). Because of their stateless nature, most functions follow the *Extract-Transform-Load* (ETL) pattern [13], meaning that the function first (*E*) reads data (e.g., an image) from a remote persistent storage service (e.g., an object store such as AWS S3), then (*T*) performs some computation (e.g., image blurring), and finally (*L*) writes the result to the remote storage.

The two main performance limitations of current FaaS platforms are well known [19, 22], namely their scheduling latency and function execution latency. The former limitation has received a lot of attention in recent years [1, 3, 8, 12, 13, 26, 27, 29]. In this paper, we focus on the latter. An acute issue here is the performance bottleneck introduced by the lack of data locality. Indeed, current FaaS platforms are typically made from two very distinct layers: a compute infrastructure and remote storage backends. This decoupling is a double

edged sword: FaaS applications benefit from unmatched elasticity but are hurt by the latency and throughput limitations of the backends for any access (*E&L* phases) to persistent or shared transient state. This problem is exacerbated in the case of function pipelines (e.g., for analytics) [9, 23] where the output of a function is the input of another, and intermediate outputs/inputs are destroyed once used, because most of the current FaaS platforms do not support direct and efficient communication between function instances [19, 22]. Prior works [9, 23] proposed to dedicate resources (providing improved storage performance) that must be booked, configured and tuned by the cloud tenants, which is at odds with the benefits expected from the serverless paradigm. Other works focusing on function pipelines have enabled direct and efficient communications between function instances, by leveraging platform-specific assumptions and features [3, 38, 46]. The Cloudburst platform [40] improves locality via per-worker data caches, which interact with a specialized storage backend using specific, coordination-free consistency protocols; to the best of our knowledge, Cloudburst requires manual/static provisioning of dedicated cache resources (esp. memory) on each worker.

In this paper we present OFC (Opportunistic FaaS Cache), an *opportunistic* RAM-based caching system to improve function execution time by reducing *E&L* latency, both for single-stage functions and function pipelines. OFC achieves this in a cost-effective manner for the cloud provider, no additional efforts (administration, code modifications) for cloud tenants, and no degradation of the data consistency and persistence guarantees. To achieve this, *OFC repurposes memory which would otherwise be wasted due to memory over-provisioning and sandbox keep alive*. Indeed, the analysis of FaaS traces (including the AWS Lambda repository) [39] reveals that users tend to over-provision the memory capacity guaranteed to their functions. Also, in order to accelerate function instance setup, FaaS platforms typically keep function sandboxes alive for several minutes (e.g., 10 in OpenWhisk and 20 in Azure) [37, 44] for serving future invocations related to the same function code. OFC opportunistically aggregates these idle memory chunks from all worker nodes to provide a distributed caching system for *E&L* phases.

This idea raises three main questions: *(Challenge #1)* How to accurately predict, at the function invocation granularity, the amount of memory needed by a sandbox (warm or cold)? *(Challenge #2)* How to build a vertically scalable caching system, with a capacity (up/down) scaling latency that is adequate even for short function executions (i.e., in the range of seconds or milliseconds) [39]? *(Challenge #3)* How to manage the caching system transparently (unmodified application code), efficiently (consistency and performance guarantees), and reliably (persistence and fault tolerance)?

For Challenge #1, we leverage machine learning (ML). Contrary to IaaS, in FaaS the provider has access to a wealth of information (such as function code, actual input parameters and runtime library), which makes ML appropriate in this context. In addition, the very high rate of invocation of a (performance sensitive) cloud function makes learning a model fast, because a training dataset is quickly accumulated. Moreover, most functions use a well-known set of common data types (e.g. multimedia), and their resource needs (especially memory) are correlated (although non-trivially) with some input parameters. OFC goes further and, for each function, extracts the main features that characterize its memory requirements and builds its prediction model. When a function is invoked, the actual memory size assigned to its sandbox is calculated using its associated ML model. Therefore, the difference between the booked memory (specified by the cloud tenant) and the predicted size is used for increasing the size of the cache on the worker node that hosts the sandbox.

For Challenge #2, the difficult aspect is the scaling down of the caching system when a worker node is lacking memory. In order to address this challenge, we use two strategies. First, we reduce the pressure on the caching system by evicting as early as possible data that are unlikely to be reused in the future. Second, we implement an optimized algorithm for object migration, allowing to keep hot objects in another worker node of the distributed cache.

For Challenge #3, we leverage ML, as well as several systems and data caching policies. First, a data item is cached only if it is likely to significantly improve the overall function execution time. To this end, OFC builds another model that outputs an estimation indicating whether or not the cache would yield improvements. Second, intermediate input/output data generated by pipelined functions are removed (but not persisted to the remote storage) from the caching system once the pipeline execution ends. Finally, to achieve the remaining goals, OFC implements several other techniques, among which: (i) asynchronous data persistence on the remote storage implemented using helper FaaS functions, (ii) adaptation of the FaaS scheduler for locality (functions are preferably run on a worker node hosting a copy of the cached data), and (iii) consistency management on the remote storage backend using "shadow objects" (i.e., placeholders for new object versions). We prototype OFC using popular software stacks: Apache OpenWhisk (OWK) [30] as the FaaS platform and OpenStack Swift [41] as the remote storage. We use RAMCloud [31] as the distributed in-memory caching system.

In summary, we make the following contributions: (i) leveraging ML to accurately predict function memory needs in a FaaS system; (ii) leveraging overbooked memory as well as sandbox keep-alive to design a cost-effective, elastic and fault-tolerant caching system; (iii) the evaluation of OFC, demonstrating that it improves by up to 82 % and 60 % respectively the execution time of single- and multi-stage FaaS applications. All our materials (source code and data sets) are publicly available at https://gitlab.com/lenapster/faascache/.

2 Background and motivations

2.1 Background

We now provide background details on Apache OpenWhisk (OWK), the FaaS framework that we leverage in our prototype implementation. Like most FaaS platforms, OWK supports polyglot users to pick a programming language of their choice (e.g., Python, NodeJS, etc.) and ways to configure trigger rules (e.g., HTTP requests to a given URL, updates within a given object storage bucket, etc.). In addition to single-stage functions, OWK provides support for function *pipelines* (a.k.a. "workflows" or "sequences"), which consist in a parallel and/or sequential composition of functions: the first function invocation is triggered by an external event and the next ones are driven by the platform, based on the completion of the function invocations and their dependency graph. Function pipelines are becoming increasingly popular for implementing massively parallel tasks (e.g., data analytics) in a simple and cost-effective manner [9, 21–23, 32].

When a function invocation is triggered, the corresponding request is forwarded to the OWK *Loadbalancer*, which is responsible for choosing a worker node that will run the function. To do so, the *Loadbalancer* maintains the current status (e.g., available resources) of all worker nodes and, using a hash of the function ID and tenant, computes the identifier (index) of the *home* worker, which is the preferred worker for handling the request. This affinity is aimed at improving code locality on the workers. Each worker node hosts a component named *Invoker*, in charge of informing the *Loadbalancer* with the current status of the node, and creating and starting function *"sandboxes"*. The latter are implemented with Docker containers in OWK.

Finally, we highlight three important aspects of sandbox management, which are also common to most FaaS platforms. First, for security, a given sandbox is never shared or reused between distinct functions or tenants. Second, a sandbox processes only a single invocation at a time. Third, to amortize start-up costs and mitigate cold-start effects, a sandbox is generally kept alive for some time (600 s in OWK) in the anticipation of future invocations of the same function.

2.2 Motivation

This section shows that worker nodes in FaaS platforms have an abundance of wasted memory capacity that can be used to build a distributed caching system cost-effectively, i.e., without the additional infrastructure required in prior works [9, 23]. We explain why machine learning is needed to reach this goal, and we also show that ETL-based functions could be a potential beneficiary of such a caching system.

2.2.1 Memory waste. Figure 1 illustrates how a sandbox uses memory during its lifetime on a worker node. In this example, the sandbox handles three events (E1–E3), which trigger three sequential invocations of the same function. We

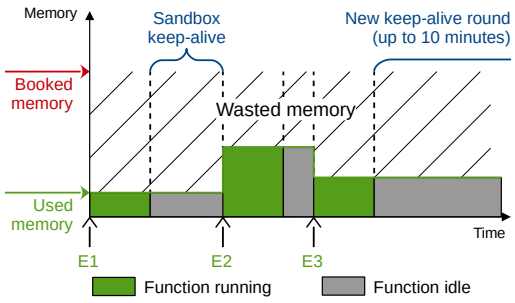

Figure 1. Timeline of a function sandbox with memory waste due to overdimensioning and the keep-alive policy.

summarize in this figure the two main sources of memory waste in FaaS platforms.

The first cause of waste is that cloud tenants often overdimension the memory resources configured for their function sandboxes. For example, a survey of AWS Lambda usage reports that 54 % of sandboxes are configured with 512 MB or more but that the average and median amounts of used memory are actually 65 MB and 29 MB [36]. We believe that this trend is mainly due to workload variations: the same function code can be triggered with different arguments and input data, which may lead to different memory needs. For illustration, Figure 2 plots the memory usage of a sample function that blurs an image, as a function of the input size and as a function of its processing-specific argument (the blurring radius). It shows that for the same function, memory usage varies widely depending on the arguments and input data, justifying resource overdimensionning practiced by cloud tenants. Hence, we believe that this trend is likely to remain, and that the resulting waste may even grow as the available range of memory configurations for sandboxes keeps increasing [6].

The second cause of waste is the sandbox keep alive policy. Indeed, to mitigate the long latency of sandbox cold starts, FaaS platforms typically keep a sandbox running for several minutes (e.g., 10 in OWK and 20 in Azure) [37, 44] after handling the first event (E1 in Figure 1) that triggered its startup. Consequently, the resources assigned to a sandbox may remain unused for long time intervals. Shahrad et al. [37] observed in Microsoft Azure Functions that "45 % *of the applications are invoked once per hour or less on average, and* 81 % *of the applications are invoked once per minute or less on average. This suggests that the cost of keeping these applications warm, relative to their total execution (billable) time, can be prohibitively high.*". The authors introduced a histogram-based solution to predict invocation frequencies and patterns for each function; this way, a sandbox can be started just before the next function invocation and shut down upon completion. Such an approach works well for some workloads but must fall back on sandbox keep-alive in various circumstances (e.g., phase changes, burstiness), which are

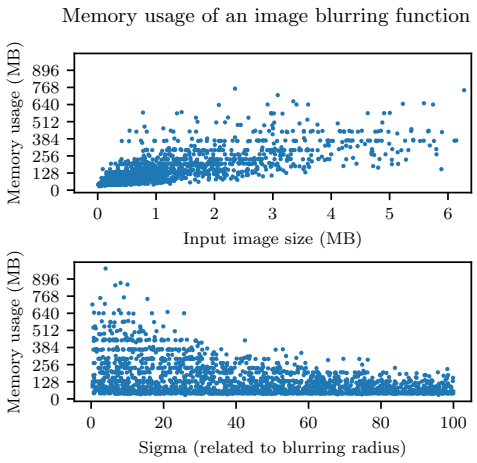

Figure 2. Illustration of the relation between a function's memory usage and two parameters: byte size of input data (top), and a function-specific argument (bottom).

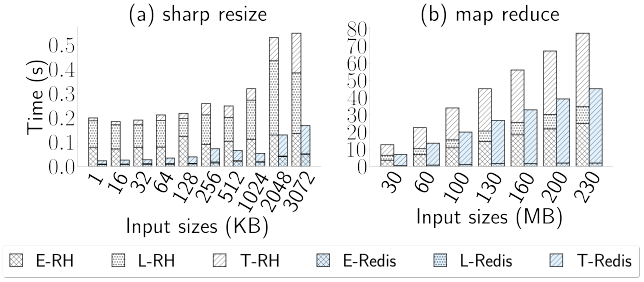

Figure 3. Duration of the ETL phases for a common image processing function (resizing) and a pipeline of data analytics functions (word count). The functions run on an OWK environment deployed on AWS EC2 using 3 compute nodes provided by `t2.medium` instances. We measure the execution times using AWS S3 as our RSDS (first bar series) and ElastiCache Redis as our IMOC (second bar series).

likely to become more prevalent as FaaS is increasingly used for more diverse and intensive applications, and also used as a means to absorb unpredictable load spikes. Besides, despite a number of recent optimizations, the overhead of cold starts is still significant: with general purpose, production-grade sandboxing technologies (e.g., containers or microVMs), a cold start latency under high load is in the range of hundreds of milliseconds [1, 26, 27], which is sensitive for very short functions and interactive/parallel applications. Consequently, some FaaS providers offer the possibility to book pre-provisioned, long-lasting sandboxes [4]. Overall, sandbox keep-alive remains an important technique for FaaS platforms for the time being, and we propose to leverage the (physical) memory waste that stems from it.

2.2.2 Memory prediction using machine learning.

In order to use the wasted memory described above, we need to estimate how much is available. In this section, we justify why we turn to machine learning for this task.

Figure 2 illustrates the complex relation between a function's arguments and input data, and its memory usage. In particular, the top figure plots memory usage against the byte size of the input data: we see that no precise correlation can be established. In other words, accurately predicting memory waste only from the byte size of the input data is not possible. Additionally, the bottom figure plots memory usage against the function's specific argument (the blurring radius): again, no precise correlation can be established from this feature alone. We have observed the same kind of trend with many other multimedia processing functions (e.g., image resizing and format conversion, speech recognition, video grayscale conversion, text summary). Furthermore, the FaaS platform has no information about a function's specific arguments: it only knows about their list and names, not about their nature, range, behavior, etc., thus adding complexity to the task of predicting the memory usage of a function.

In summary, predicting memory waste for a function invocation requires to take into account uncharacterized function-specific arguments in addition to features of the input data. As shown in the remainder of this paper, machine learning (ML) can manage this complexity without prior knowledge.

2.2.3 Remote shared data store latency.

We measure the impact of the systematic utilization of a remote shared data storage (hereafter RSDS) imposed by the FaaS model to ETL-based functions. To this end, we run in AWS example single-stage functions (image processing) and example multi-stage functions (analytics). We use AWS S3 as the RSDS and we experiment with various input sizes. Figure 3a, first bar series, presents the contribution of each ETL phase for one image processing function (`sharp_resize`). For instance, *E&L* represents up to 97 % of the total execution time for a 128 kB image size. Figure 3b, first bar series, presents results for MapReduce word count. *E&L* represents up to 52 % of total execution time for a 30 MB input text file.

A typical way [9] to address this bottleneck is to use an in-memory object cache (IMOC) such as Redis between the cloud functions and the RSDS. The second bar series in Figures 3a and 3b show the results of the same experiments as above when S3 is replaced with Redis. We can see that the contribution of *E&L* becomes negligible. However, the utilization of such an IMOC-based solution is not without downsides for FaaS tenants. They must explicitly allocate and manage IMOC resources. In addition, they must modify their function's code (or introduce a library) to interact with the IMOC layer and also to address potential consistency issues. All these constraints and extra burden are at odds with FaaS principles. OFC, the system that we propose, achieves

performance benefits comparable to an IMOC-based solution but without the aforementioned limitations.

3 Design assumptions

Our design principles are not tied to the specific components used in our prototype (OWK and Swift) but they do rely on a small number of assumptions, which we mention below. Overall, we assume a fail-stop model (with details similar to RAMCloud [31]).

For the FaaS infrastructure, we only assume two characteristics, which are very common in current platforms: (i) the use of a sandbox keep-alive strategy (either via a simple idle timeout like in OWK and AWS Lambda, or through a control-loop approach like in Kubernetes-based platforms) and (ii) the use of a (per-function) central component to dispatch invocation requests to the sandboxes (like the controller component of OWK or the scheduler component of Kubernetes-based platforms).

For the remote storage system, we do not make any major assumption. First of all, we are agnostic to the data abstraction level (e.g., file-based, object-based or key-value interface). We simply assume the possibility to register handlers, to be triggered upon the invocation of certain operations. In addition, we aim at supporting storage systems with various consistency guarantees including strong ones, such as linearizability. We believe that this is important because this simplifies the work of applications developers (a number of object storage systems are now evolving towards stronger consistency semantics [5, 17]) and a strongly consistent storage backend also simplifies the design of hybrid applications combining FaaS and other services (e.g., Infrastructure as a Service). For caching, we assume that the remote storage supports transparent interposition. We store full copies of the object data in the cache, and we focus on small objects (10 MB or less) because they benefit the most from caching (as shown in §2.2.3). Note that some of the workloads that we study use large data sets (hundreds of megabytes) but that the corresponding input, intermediate and output data are actually split into many small objects.

Finally, we assume that the ML infrastructure has access in clear text to the function invocation arguments and also to the object's metadata. We leave the support of black-box/encrypted inputs to future work.

4 OFC overview

Figure 4 presents the architecture of OFC. Components that we add to OWK in order to provide OFC are indicated by color-filled boxes in the diagram and "*new*" in our description. When OWK's *Controller* receives a function invocation request, it first asks the (*new*) *Predictor* to predict the amount of memory (noted M_p) that should be assigned to the sandbox (details in §5.1). The *Predictor* also returns a boolean (noted *shouldBeCached*) indicating whether it is beneficial

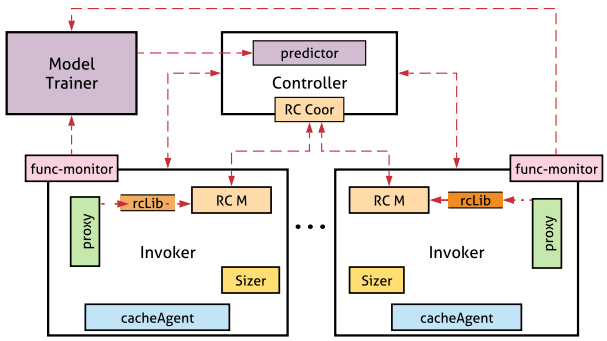

Figure 4. OFC architecture overview. All color-filled boxes are new components that we add to OWK.

(i.e., will lead to a significant decrease in execution time) to cache the data objects read/written during the function invocation (§5.2). Based on M_p, the *Controller* then selects the invoker (worker) node that will handle the function invocation request (§6.5). OFC modifies the native selection algorithm, by taking into account (i) the amount of memory currently provisioned in the existing and idle sandboxes for the same function (if any) and (ii) data locality (an invoker node holding the master/in-memory copy of an object in its cache instance is prioritized). In addition to the default information sent by the *Controller* to the selected invoker node, the request also includes M_p and *shouldBeCached*. Upon receiving this information, the invoker node, via the (*new*) *Sizer* component, (re)sizes the sandbox.[1] Finally, the (*new*) *CacheAgent* resizes (down/up) the caching storage (§6.4) depending on the new contribution of the function to the cache.

Read and write operations performed by a function to the RSDS are transparently captured at execution time by the (*new*) *Proxy* component, which uses our (*new*) `rclib` library to redirect them to the caching system. To provide the basic functionalities of the latter, OFC relies on RAMCloud [31] (components with *RC* prefix in the figure), a RAM-based durable key-value store (§6.1). For write operations, the caching storage is only solicited when *shouldBeCached* is true (§6.3 for the caching policies). In this case, *Proxy* injects in the FaaS platform a *Persistor* function to asynchronously persist the cached data to the RSDS (§6.2).

To take into account false predictions of M_p (lower than the actually needed memory), the execution of each function is locally monitored by a (*new*) *Monitor* component (§5.3). The latter has two goals. First, it can ask the *Sizer* to quickly increase the memory capacity of a sandbox when the latter lacks memory. Second, after the completion of every function invocation, it sends the (maximum) amount of memory consumed to the (*new*) *ModelTrainer* component. The latter

[1] On a cold start, the invoker will create the new container with this memory constraint; on a warm start, it updates the memory constraint of the existing container selected to run the invocation.

periodically retrains all memory prediction models using these numbers and updates the *Predictor*.

Overall, our prototype implementation requires the following volumes of new or modified lines of code: 5 kLoC for OWK (7.5 % of the original code base) including the ML part, 10 kLoC for RAMCloud (6 %), 15 LoC for Swift (0.3 %).

5 ML modules

The *Predictor* and the *ModelTrainer* work together to give predictions on two topics: (i) the memory requirements of a function invocation, and (ii) the performance improvement that the latter can achieve with the cache.

5.1 Prediction of physical memory requirements

The *Predictor* performs on a per-invocation basis: using the function's memory model, learned by the *ModelTrainer* through the function's lifetime. It takes as input the parameters of the function invocation request, and outputs the predicted memory requirement of the invocation. We store all the function models in OWK's database (CouchDB), so when a function is invoked and OWK fetches its metadata, it also gets its model to be used by the *Predictor*.

5.1.1 ML algorithm. Our choice of the ML algorithm, and the features considered for modeling, is guided by the following constraints, on which we expand below: model update, model output, model inputs, and prediction speed.

Model update. We must be able to update the model throughout a function's lifetime for two reasons: first, the model is blank when the function is uploaded to the FaaS platform, and second, once the model is in use, it must be corrected in case of bad predictions. This warrants either for an algorithm that accepts incremental updates, or for maintaining a training dataset that is updated with more exhaustive data before retraining the model.

Model output. We loosely defined the model's goal as "predicting the amount of memory", but actually we settle with predicting a *range*. Indeed, OWK defines a range of permitted memory allocations, ($[0, 2]$GB by default). We divide this into intervals in order to formulate the model as a *classifier*, making it easier to do predictions, as commonly practiced [10]. Hence, the amount of memory to allocate is the upper bound of the predicted interval. We discuss the size and number of classification intervals in §7.1.

Model inputs and specificity. The input to the model can only include features that are readily available from an incoming request to the FaaS platform. For instance, a function that blurs images does not expect the same arguments as a function that compresses audio; because the nature of function inputs (image, audio, video, etc.) vary, so do the features that can be extracted from a request as input to the model. Thus, we must learn one model per function, that is capable of handling function-specific arguments.

Prediction speed. The memory prediction intervenes on the critical path of a function invocation, which must guarantee low latencies. Wang et al. [44] measured median cold start latencies for various FaaS providers in the order of 100 ms, and median warm start latencies in the order of 10 ms. Thus, we set the target prediction time at 1 ms.

Based on these criteria we choose *decision trees*: they can be seen as a cache lookup operation, where the *Predictor* looks up the amount of memory used previously for a given set of features [42]. Moreover, decision trees are fast at classifying. We elect to use the J48 decision tree algorithm (a Java implementation of C4.5 [35]) with 16 MB intervals from 0 to 2 GB. Section 7.1 shows the results validating our choices.

5.1.2 Feature selection and processing. We select features depending on the function's input type: image, audio, and video. When applying ML to such inputs, the usual goal is to *characterize the content*, e.g., identifying shapes in an image. However, this is not the case here: our models do not learn the content of the processed media, but rather the descriptive features available in the function request parameters or media metadata, which may affect memory usage.

J48 does not require pre-processing the features, but we avoid extracting them on the critical (invocation) path whenever possible. We leverage the fact that in our situation, all data objects reside in the RSDS: we extract the features when an object is created, and store them alongside it, as a background task. The only case in which we perform the extraction synchronously is when a function invocation stems from a storage trigger (object creation or update). We define a set of common input features, e.g., input file size is used for all functions, while image processing functions include pixel dimensions, and audio/video functions include duration, etc. Further, function-specific arguments are also used as input features (for instance, an image blurring function would receive, along with the image itself, a blurring radius). The evaluation results in §7.1 demonstrate that these feature sets are sufficient to make accurate predictions.

A FaaS platform such as OWK has the knowledge of the list and values of arguments sent to a function, so it is easy to extract them to be used as features. However, no semantic information is known about the arguments themselves. This raises two challenges. First, how to identify the function arguments that correspond to object identifiers (needed for feature extraction, as mentioned above)? In the case of functions triggered by object creations, the target objects are determined automatically; otherwise, we rely on manual annotation of the function arguments. Second, regarding the remaining arguments, how to feed the ML algorithm with their opaque values? (Are they floats? What is their range? Are they nominal/discrete values?) A benefit of decision trees

such as J48 is that this information is mostly not required; only for nominal values do we need to learn their ensemble. This is easily done because we actually keep a training set of invocations in order to update the model when necessary (see §5.3), so we can have an exhaustive view of all the nominal values that the function ever received.

5.2 Caching benefit prediction

We state that caching is beneficial for a given invocation when the (wall clock) time taken to extract and load the data dominates the total execution time. This is expressed as the ratio $\frac{T_e+T_l}{T_e+T_t+T_l}$ being greater than 0.5; with T_e, T_t, and T_l corresponding respectively to the time taken by the extract, transform, and load phases. In this case, the *Predictor* is a binary classifier that outputs a boolean telling if caching is useful. Learning this information is similar to predicting memory intervals: we also use J48, with the same features, learning one model per function.

5.3 Managing prediction errors

5.3.1 Memory prediction. Underprediction has negative effects on the invocation: it may experience swapping activity, resulting in degraded performance, or even abrupt termination by the *Out-Of-Memory (OOM) killer* daemon of the Linux host.

As a first step to avoid this situation, the memory predictions are not actually used until the ML model is accurate enough, which we define as a *maturation criterion*.

Definition. Let C_k be the k-th classification interval; a greater k corresponds to a higher amount of predicted memory, and k^* is the index of the true interval.
The maturation criterion is:

- 90 % exact-or-overpredictions (*EO-predictions*): the model predicts C_k with $k \geq k^*$ for 90 % of the cases;
- 50 % of underpredictions are within one interval of C_{k^*}: when the model predicts C_k with $k < k^*$, we have $k = k^* - 1$ for at least 50 % of the cases.

Once the predictor meets these requirements, we further mitigate the underprediction problem in three ways. First, we conservatively use the next greater interval as the predicted memory amount, which ties into the criterion of "50 % of underpredictions are within one interval of the correct prediction" described above. By doing so, 50 % of underpredictions become exact predictions, and we ensure that we have $(0.9 + 0.1 \times 0.5) = 95$ % EO-predictions. Second, if an invocation fails because of the OOM killer, it is immediately retried with the memory limit raised to the amount set by the tenant. Third, OFC also monitors invocations during their execution to measure the actual memory usage (by periodically reading statistics from `cgroup`, the facility used by Docker). Whenever a problem of memory exhaustion is detected, the model is corrected quickly to take into account this error for future invocations under the same conditions.

In addition, OFC also attempts to dynamically detect sandboxes with high memory pressure and dynamically raise their memory cap. We enable this approach only for invocations that have run for at least 3 s. Indeed, shorter invocations are frequent (50 % of the invocations in the study of Shahrad et al. [37]) and unlikely to be affected by under-predictions for memory sizing. Hence, we avoid the monitoring overheads in the case of short invocations.

5.3.2 Caching benefit prediction. An error from this model will not degrade performance compared to a setup without a cache. If the cache is predicted useless but could have been useful (false negative), there is no performance degradation, only a lost opportunity; and in the event that the cache is predicted useful but ends up useless (false positive), it only puts a slight overhead on the *CacheAgent* component.

5.3.3 Retraining. For both models, prediction errors are corrected after the fact by periodically updating them. Given that J48 is not an incremental model, the *ModelTrainer* needs to fully re-train the models when new data is available. We make this practical by maintaining a small, but valuable training dataset: after the Predictor maturation criterion described above is reached, we only add data about invocations for which the memory model predicted an interval that was too low, or extremely too high (the model predicted C_k with $k - k^* > 6$). We also give a higher weight to the training data about underprediction cases in order to better avoid them.

6 Cache design

This section details how OFC implements its caching system in OWK, and how it is managed and used.

6.1 Cache storage

Our infrastructure leverages the RAMCloud [31] distributed key-value store (with data partitioning and replication) for the management of the cached data. More precisely, in our design, each machine running an OWK Invoker also hosts an instance of a RAMCloud *storage server* (which comprises two components: a *master* and a *backup*; the former manages the in-memory storage of the primary copy for some of the objects and the latter handles the on-disk storage for the backup copies of other objects). The storage capacity of RAMCloud is dynamically adjusted, both horizontally and vertically. Unlike in a vanilla RAMCloud setup, OFC allocates only a fraction of an Invoker machine's resources to a storage server; this fraction depends on the memory booked but left unused by functions. Section 6.4 describes the scaling process of each server instance.

We chose to use RAMCloud for four main reasons: (i) it is specifically aimed at aggregating the (main memory and disk) capacity of the cluster nodes, (ii) it achieves very low latency, (iii) it provides strong consistency and fault tolerance guarantees, and (iv) it ensures durability and efficient RAM

usage (backup copies are stored on disk rather than RAM). Besides, RAMCloud is optimized for storing small data objects, which is in line with the object sizes that benefit the most from the cache,[2] for the workloads that we consider (see §2.2.3). We leave for future work the (efficient) support for arbitrary object sizes.

Regarding fault tolerance, the cache mainly relies on the support provided by RAMCloud (replication and fast recovery) and OWK (retries of failed/timed-out invocations). The cache is transparent regarding the fault tolerance model to be considered by application developers (functions are expected to have idempotent side effects).

6.2 Persistence and consistency

Given our objective of transparency, the caching layer introduced by OFC must not degrade the consistency and persistence guarantees offered by the RSDS.[3] This section describes how we achieve this goal. In a second part, we then explain when and how these constraints can be relaxed in order to improve performance.

To keep the RSDS up to date, OFC must synchronously forward write requests (i.e., regarding a `create`, `update` or `delete` operation for an object) to the RSDS. The rcLib uses the following approach in order to achieve better performance: the synchronous request issued to the object store contains an empty payload and is used to create a placeholder (hereafter named *"shadow"*) for the newly created/updated object *Obj*. It is associated with a set of metadata tags (both in the cache and in the RSDS): two version numbers, respectively for the latest version of *Obj* and the latest version available in the RSDS (a discrepancy between the two indicates that the RSDS does not store *Obj*'s current data payload). Once the synchronous RSDS request has completed and the write has been persistently stored in RAMCloud, the rcLib acknowledges the request to the client application (function) and schedules the *persistor*, a background task running as a (FaaS) function. The *persistor* code consists in (i) pushing *Obj*'s payload from the cache (RAMCloud) to the object store and (ii) update its metadata. The version numbers are also used by *persistor* tasks to enforce that successive updates to the same object are (asynchronously) propagated in the correct order to the RSDS. Our experiments show that this mechanism, akin to write-back, is always beneficial even for small payloads, and thus is always used for cached objects.

The notion of shadow object is also useful to provide strong consistency guarantees when a client application directly issues a request to the RSDS (e.g., typically, a non-FaaS application). Here, we leverage the support for webhooks provided by Swift: a callback function is registered and triggered upon each read request. The webhook checks if the RSDS holds the latest version of the object (by comparing the values of the two above-described version numbers). If this is not the case, the webhook notifies the OWK controller so that the latter can boost the scheduling of the corresponding *persistor* task. The webhook only terminates (and allows the completion of the external read request) once the latest data payload is available in the RSDS. Similarly, if an external client issues a write request while the cache holds a copy of the object, a webhook is used to (synchronously) invalidate the cached copy in RAMCloud before performing the operation on the RSDS. Besides, in the case of several function invocations performing (concurrent or serial) accesses to a cached object, strong consistency is enforced by RAMCloud. RAMCloud provides linearizable semantics for failure-free scenarios and strongly-consistent "at-least-once" semantics otherwise [31], and can be extended to support full linearizability and multi-object transactions [24].

While the above-described techniques (synchronous write requests, persistors and webhooks) are useful to provide full transparency, we observe that they are not always necessary in practice. Indeed, in many FaaS use cases, most or even all of the accesses to the object store are mediated through the FaaS code. Therefore, our system allows tenants to disable the above-mentioned facilities (via metadata tags and settings, on the scale of each bucket/object/account) in order to improve performance. In such a case, the consistency between the cache and the object store is relaxed (writes are only propagated lazily to the object store, upon the cache eviction decisions discussed in §6.3) and persistence relies on the (on-disk) replication provided by RAMCloud.

6.3 Caching policy

To improve cache usage for the functions that will benefit the most from it, OFC relies on the following heuristics for admitting objects in the cache and evicting them.

For a given invocation of function `F`, an object is considered for caching only if it satisfies two conditions. First, it must be smaller than the maximum object size allowed in the cache; we use 10 MB in our prototype, according to our cache efficiency characterization (see §2.2.3). Second, as explained in §5.2, the predicted performance benefits of the cache for `F` and the corresponding object(s) must be significant. Furthermore, in the case of a pipeline, the output objects produced by the intermediate stages (functions) of the pipeline are removed from the cache when the last function of the pipeline has completed. In addition to the previous policies, final output objects (i.e., produced at the end of a pipeline or by a single-stage function) are discarded from the cache as soon as they have been written back to the remote storage.

In addition, to reclaim more space, the cacheAgent periodically evicts objects that have not been recently accessed.

[2] By default, the maximum object size in RAMCloud is 1 MB. We extended it to 10 MB based on our observations.

[3] Some object storage systems (like Swift and AWS S3) do not provide very strong consistency guarantees such as linearizability. In such a case, client applications must typically avoid concurrent accesses to mutable objects or rely on an external synchronization facility. In our work, we assume that applications are designed according to these guidelines if needed.

We extended RAMCloud to maintain, for each object, a read access counter n_{access} and a timestamp T_{access} that records the epoch of the last access. In our current setup (tuned empirically), this periodic eviction is triggered every 300 s, and the eviction criteria are: $n_{access} < 5$ or $T_{access} > 30$ min.

6.4 Autoscaling

The horizontal scaling (in/out) of OFC relies on the support provided by OWK and RAMCloud. Below, we mostly focus on how OFC supports vertical scaling. OFC opportunistically hoards the unused (but already booked) memory on each Invoker node. Within an Invoker node, workload variations introduce two main challenges regarding this aspect. First, given that the memory consumption of most functions is input-sensitive, a sandbox may have widely fluctuating memory requirements during its lifetime (recall that a sandbox may serve multiple invocations of the same function). Second, unexpected load spikes may require to quickly release some (or even all) of the cache resources in order to accommodate more demanding requests and/or a greater number of sandboxes. Our design is impacted by three quantitative aspects. The first aspect is the end-to-end time needed to process an empty function throughout the (distributed) OWK infrastructure, which is in the range of 8 ms. The second aspect is the time required to dynamically reconfigure (i.e., scale up or down) the memory pool of a RAMCloud instance, which is in the range of dozens of milliseconds, as shown in §7.2.1. The third aspect is the time taken to adjust the resource limits of a sandbox (in OWK, which uses Docker, this is a syscall to the `cgroup` Linux subsystem), which is in the range of 24 ms.

To address the first challenge, we adjust the memory of a sandbox for each invocation: scaling up the memory resources of a sandbox involves scaling down the ones of OFC, and vice versa. We optimize the critical path by executing all the memory capacity adjustements asynchronously: the function invocation is processed before the completion of the memory resizing operations (`cgroup` syscall for the sandbox and RAMCloud control request). Yet, in the case of a sandbox capacity scale-up, this may introduce the risk of memory capacity violation, leading to the failure of the function invocation (which implies retrying the invocation, and leads to increased completion times and waste). This risk is exacerbated by potential memory under-predictions and bursty workloads. To mitigate the occurrences of such events, each Invoker node provisions a slack pool of memory, whose size (initially 100 MB) is adjusted every 120 s based on an estimation by sliding window, of the local memory churn (measured every 60 s).

To address the second challenge of fast reclamation of the cache resources, we use the following decentralized approach. The `cacheAgent` on an Invoker node must choose and release objects from the local cache instance. It first selects the output objects (in the case of function pipelines, final outputs) that have been persisted on the RSDS but not yet discarded locally. If more space is required, the `cacheAgent` proceeds with input objects and evicts them on an LRU basis (until enough space is available). In parallel, it also triggers the write-back of the dirty output objects and discards them upon completion. The `cacheAgent` attempts to keep the hot input objects in the cache by offloading their master (in-memory) copy to another RAMCloud storage node. To achieve this, we do not rely on the standard object migration protocol supported by RAMCloud (which systematically sends the target object to the destination node); instead, we use the following optimized approach to speed up the migration. For each object O chosen for eviction on a node M_{old}, a new master node M_{new} is elected among the backup nodes (i.e., holding an on-disk copy of O). O is then loaded in the memory of M_{new}, and M_{old} removes it from main memory (but becomes a backup and keeps an on-disk copy). This way, no inter-node transfer of O is necessary. By doing so, OFC ensures high availability of the remaining cached objects and maintains the required replication factor for fault tolerance.

6.5 Request routing

We aim at (i) achieving good load balancing between the invoker nodes (regarding the load incurred by the function invocations but also by the caching service), (ii) limiting the cache management overheads (e.g., memory resources adjustments and transfers of cached objects between nodes), and (iii) improving data locality. To this end, we modify the policy used by OWK's *Loadbalancer* component (see §2.1) to route function invocation requests. Similar to the original design, a request for a function `F` is always routed to an idle (warm) sandbox set up for `F` if there is one (to avoid cold starts), and otherwise, a new sandbox is immediately created (to avoid queueing latency behind long-running requests). If a new sandbox must be created, the target Invoker node is preferably the one currently hosting the master (in-memory) cached copy in its local RAMCloud storage instance (if it exists and has sufficient resources). To find such a node, the controller parses the function invocation request (to extract the object ID among the arguments) and queries the RAMCloud coordinator. If there are multiple available sandboxes, the routing algorithm uses the following criteria, by decreasing order of priority: (i) the difference between the current memory capacity of the sandbox and the predicted capacity for the new invocation (smallest difference is preferred); (ii) the available memory capacity on the Invoker node (if the capacity must grow); (iii) the locality of the data (sandboxes co-located with the requested object are preferred); (iv) the idle time of the sandbox (more recently used sandboxes are preferred, so that the older ones can eventually time out and be reclaimed if they are in surplus).

7 Evaluation

This section presents the evaluation results of OFC.

Evaluation goals and methodology. We evaluate the following aspects: (i) concerning the ML module, the accuracy of the prediction model, the prediction time, and the model maturation quickness; and, (ii) concerning the caching system, the overall performance gain and costs.

The testbed is composed of 6 physical machines, interconnected via a 10Gb/s Ethernet switch running Ubuntu 16.04.7 LTS. The hardware characteristics are as follows: 2 Intel Xeon E5-2698v4 CPUs (20 cores/CPU), 512GB of RAM, an Intel Ethernet 10G 2P X520 Adapter, and a 480GB SSD. We use one machine to host all the OFC controllers (*Model Trainer* and *Controller* boxes in Figure 4). Another machine is dedicated to the storage system. The remaining machines are FaaS worker nodes.

Benchmarks. We evaluate single- and multi-stage functions. For the former, we use 19 multimedia processing functions, available online (see our code repository in §1). For multi-stage functions, we study four applications: two data analytics applications as in [23] (a MapReduce-based "word count" application; Thousand Island Scanner (THIS) [33], as well as (in §7.2), a distributed video processing benchmark), a cloud-based Illegitimate Mobile App Detector (IMAD) application [45][4], and an image thumbnail generator pipeline (Image Processing) from the ServerlessBench suite [47].

We run each experiment 5 times and report the average results.

7.1 ML model evaluation

7.1.1 Accuracy.

We first evaluate the accuracy of ML predictions concerning memory requirements, with various decision tree algorithms: J48 (an implementation of C4.5 [35]), RandomForest [7], RandomTree [34] and HoeffdingTree [20]. Then, we discuss the results regarding the prediction of caching benefits.

Prediction of physical memory requirements. We use cross-validation to prevent overfitting. Moreover, we experiment with $n = \{64, 128, 256\}$ intervals, with respective interval sizes of $\{32, 16, 8\}$MB. Table 1 shows evaluation results. It demonstrates that *J48 and RandomForest are the most accurate algorithms*: with 16 MB intervals, they achieve more than 80 % accuracy, and more than 90 % accuracy on EO-predictions; remember that we are interested in EO-predictions because it is a component of the maturation criterion, as explained in §5.3. An interval size of 16 MB also allows keeping prediction times very low, as shown further below. Ultimately, we elect to use J48 because its prediction time is much shorter than RandomForest's (see §7.1.2).

Our results also demonstrate that overpredictions are not a problem because, as shown in Figure 5, they remain close to the correct value: 90 % of them are within 3 intervals of the correct one, resulting in an average memory waste of only 26.8 MB with 16 MB intervals. In any case, EO-predictions are always favored over underpredictions. Indeed, as explained in §5.3, we use the next greater interval; Figure 5 does not reflect this behavior and shows raw predictions.

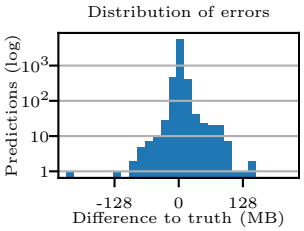

Figure 5. Errors in memory requirements prediction using J48, with 16 MB intervals (all functions combined).

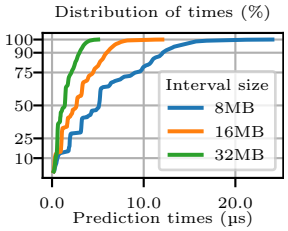

Figure 6. Time for memory requirements prediction, using J48 and varying interval sizes (all functions).

Table 1. Evaluation of ML algorithms with varying interval sizes. Results are fractions of exact, and exact-or-over predictions, averaged over all functions.

Interval size	Algorithm	Exact (%)	Exact-or-over (%)
32 MB	HoeffdingTree	81.09	87.65
	J48	91.27	95.77
	RandomForest	92.66	96.20
	RandomTree	89.84	94.23
16 MB	HoeffdingTree	72.01	84.81
	J48	83.35	92.73
	RandomForest	84.82	92.76
	RandomTree	79.23	88.69
8 MB	HoeffdingTree	63.40	79.17
	J48	75.88	87.91
	RandomForest	78.17	89.42
	RandomTree	72.27	84.12

Prediction of cache benefit. Here, we validate our choice of J48 to predict caching benefit (as defined in §5.2). The *precision* of the model is 98.8 % ; a higher precision means the model is more often correct when predicting that the cache is useful. Its *recall* is 98.6 % ; a higher recall means the model detects more exhaustively the cases in which the cache is useful. The F-measure — the harmonic mean of precision and recall, used as a global efficiency score — is 98.7 %. These results outperform the other classifiers, so we find J48 is a good fit to predict cache benefit.

7.1.2 Prediction speed.

We evaluate the classification speed, i.e., the time taken for a single prediction, for our two models. Results are shown in Figure 6. Over all functions, and with a memory interval size of 16 MB, the median memory requirements prediction time is 3.19 µs, and the 99th percentile is 12.54 µs. The classification speed for the cache benefit model is similar to the speed

[4] We reimplemented IMAD as a sequence of serverless functions.

of predicting memory usage, so overall the global prediction speed remains negligible. For reference, the prediction time using RandomForest (an algorithm that produces prediction results of quality similar to J48) is 106.29 µs at the median, and the 99th percentile is 173.05 µs.

7.1.3 Model maturation quickness.

We also checked the model maturation quickness, i.e., the number of training inputs that the model needs to learn from, in order to be accurate enough, as defined by the two criteria from §5.3. Only the maturation quickness of the model that predicts memory is evaluated, because using the model that predicts cache benefits is subordinated to using the former — and prediction errors of the latter are not problematic. Remember that we are in a context where the model is learned over time, so the number of training inputs that represents the quickness is actually a number of invocations of the model's function. We start checking the maturity after 100 invocations, so this is a minimum. In our evaluation, the median maturation quickness is 100 invocations; this result includes 11 of 19 functions that matured in 100 invocations or less. 75 % of the functions matured with less than 250 invocations, and 95 % did so under 450 invocations. As an illustration, Shahrad et al. [37] state that 99.6 % of the functions they study are invoked at least once per minute, so 95 % of them would mature in less than 450 min (7 h 30 min).

7.2 Cache performance evaluation

We perform two types of experiments. We first evaluate OFC while running a single function. We then evaluate OFC while running several sandboxes concurrently, for diverse function invocations. We compare against two alternatives based on standard OWK: OWK with all data stored in the Redis in-memory cache (noted OWK-Redis), or OWK with all data in the Swift persistent RSDS (OWK-Swift). These baselines represent the best and worst-case data access time respectively.

7.2.1 Micro evaluations.

Benefits of OFC's cache. We first run each of our multimedia and data analytics functions alone using OFC and our two baseline configurations. We do this while varying the input data size and compare the end-to-end latency. Regarding OFC, we evaluate three scenarios for fairness: (*LH — LocalHit*) the input data is in OFC's cache, on the same worker node that runs the function, (*M — Miss*) the input data is not in the cache, and (*RH — RemoteHit*) the input data is in the cache but on a different worker node. Recall that in OFC, outputs are always buffered (i.e., stored in RAMCloud in write-back mode) regardless of the scenario, which helps multi-stage functions.

We present results for 6 single-stage functions and the 4 multi-stage pipelines, shown in Figure 7. Each stacked bar in the figure shows the time for the Extract, Load, and Transform phases bottom to top. In scenario *LH*, OFC outperforms

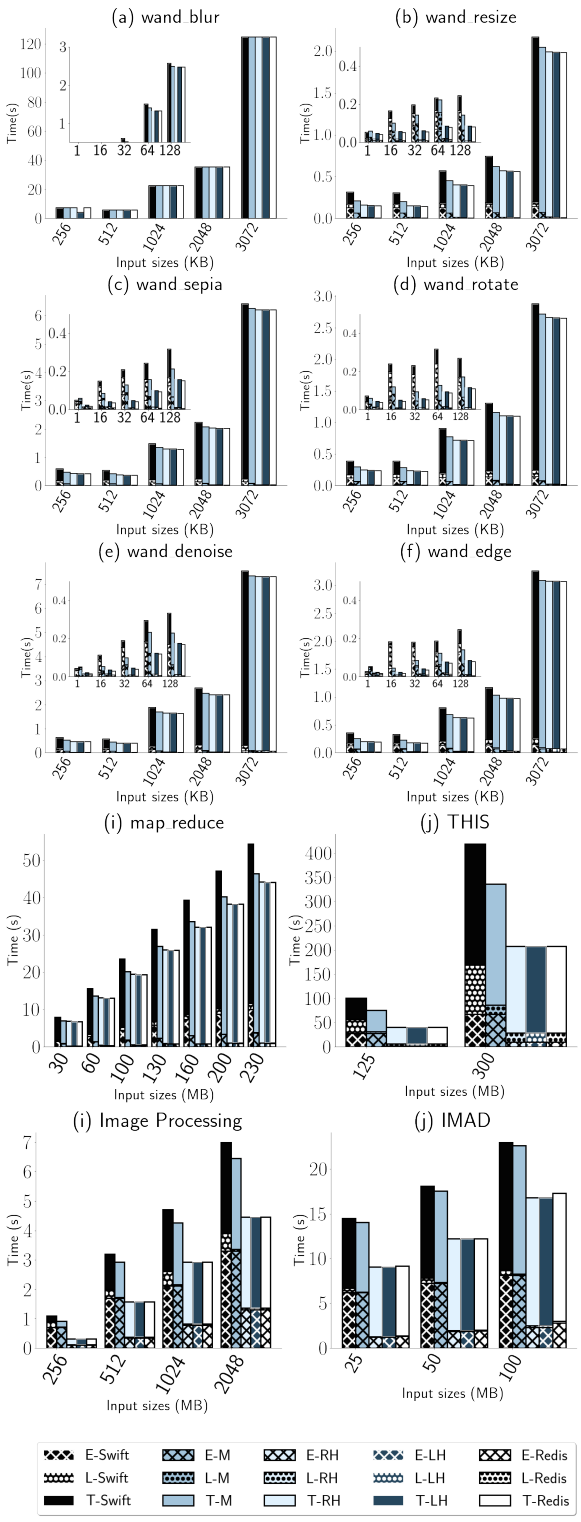

Figure 7. Duration of ETL phases for 6 common image processing functions and 4 multi-stage data analytics functions (MapReduce word count, THIS, IMAD, and ServerlessBench - Image Processing). We compare OWK-Swift, OWK-Redis and OFC (under scenarios *LH*, *M*, and *RH*).

OWK-swift by up to ≈82 % for single-stage functions (180 ms down to 32 ms for the wand_edge function with 16 kB input) and up to ≈60 % for multi-stage functions (105 s down to 35.84 s for THIS with 125 MB input). In absolute numbers, OFC reduces the latency of wand_edge by a total of 150 ms (with respectively 42 and 108 ms of savings for the Extract and Load phase). Overall, OFC achieves very close results w.r.t. OWK-Redis (with maximum differences ranging from −3 % to 2 % in completion times). We attribute these minor differences to two main sources: (i) design and implementation differences between the two caches (Redis and RAMCloud) and (ii) in the case of small requests, the overhead of the *Predictor* and *Sizer* components of OFC, which adds about 6 ms of latency.

In scenario *M*, the difference between OFC and OWK-Redis is more pronounced since all initial accesses must go to Swift. OWK-Redis outperforms OFC by up to 65 % for single-stage functions and up to 46 % (336.25 s to 207.48 s for THIS with 300 MB input) for multi-stage functions. However, OFC still outperforms OWK-swift in this scenario by up to 75 % for single-stage functions and up to 24 % (see wand_edge function with 16 kB input data) for multi-stage functions (see THIS with 125 MB input data). This is explained by the fact that although the input data comes from the RSDS in OFC, outputs are always cached i.e., (the Load phase is improved).

The results of scenario *RH* show that retrieving cached data from a remote worker node still provides a significant speedup compared to reading from Swift. Compared to scenario *LH*, remote access increases execution time by up to 12.76 % for single-stage functions (19.6 ms up to 22.1 ms for the wand_denoise function with 1 kB input) and up to 0.85 % for multi-stage functions (6.52 s up to 6.57 s for map_reduce with 30 MB input).

In all OFC scenarios, the time needed to persist a shadow object to the RSDS in the *Load* phase is constant (about 11 ms) since its size is independent of the output size.

Potential negative impact. OFC's cache can introduce a delay on function setup when the worker node lacks memory. We only show the results for wand_sepia as the observations are the same for other functions. We evaluate four scenarios regarding the status of the worker node. In the first scenario, Sc_1, shrinking the cache does not involve data migration/eviction. In the second scenario, Sc_2, shrinking the cache requires data migration. In the third scenario, Sc_3, shrinking requires eviction without migration. We compare these scenarios with the baseline, noted Sc_b, where the execution of the function does not require any cache shrinking. In the scenarios Sc_{1-3}, the current memory size of the (warm) container is 64 MB (the smallest configurable memory in OWK). We consider input data sizes ranging from 1 kB to 3072 kB, which result in function memory requirements between 84 MB and 152 MB.

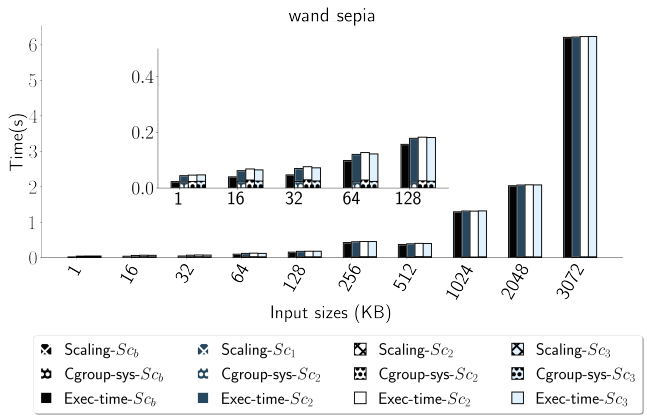

Figure 8. Impact of OFC's cache scaling on the wand_sepia function's latency. For each scenario, we plot the scaling time, the time for the container memory limit update (noted *cgroup syscall*), and the overall function execution time.

Figure 8 presents, for each scenario, the following metrics: the time needed to scale down OFC's cache, the time needed to increase the container's cgroup memory (noted *cgroup-sys*), and the overall function execution time. The time *cgroup-sys* phase is constant, in average 23.8 ms (0.8 s for the cgroup syscall and the remainder being the docker update command.). The scaling time in Sc_1 as well as in Sc_3 is also constant, in average 289 µs and 373 µs respectively. It varies in Sc_2 according to the aggregated size of the migrated objects. For this experiment, the migration times range from 401 µs (20 MB to migrate) to 2.2 ms (88 MB). More generally, we measure migration times of 0.18 ms for 8 MB, 1.2 ms for 64 MB, 3.8 ms for 256 MB, 7.5 ms for 512 MB and 13.5 ms for 1 GB. In the worst case (1 kB input size), OFC's cache scaling (*cache shrink+cgroup syscall*) takes 24.3 ms in total, which represents a 50.4 % overhead on the overall execution time (48.2 ms). We believe that this scenario is likely to be rare w.r.t. our data caching policy, which tries to free as early as possible the unused data from the cache.

7.2.2 Macro-experiments.

Methodology. OFC's efficiency depends on the workload characteristics. We built FaaSLoad, a load injector for OWK, which allows emulating several tenants with different loads. In this experiment, we consider one FaaS function per tenant. Overall, FaaSLoad prepares the input data (in the RSDS) for the invocations of each function, then performs the function invocations at different intervals within a given observation period. The invocation interval can be configured as periodic or based on the exponential law.

We set up 8 tenants and associate each of them with a distinct function from Figure 7. Our RSDS is Swift. We run FaaSLoad for 30 minutes. Functions invocation intervals follow the exponential law with $\lambda = \frac{1}{60}$, corresponding to a mean invocation interval of 1 minute. We consider three

different profiles of cloud tenants, which use distinct approaches to configure the memory size of their functions: (i) *naive*, i.e., always reserving the maximum memory size allowed by OWK (2 GB); (ii) *advanced*, i.e., reserving the maximum amount of memory that has been used by a function (according to the previous runs); (iii) *normal*, i.e., reserving 1.7x the memory size chosen by an advanced tenant, a common situation in practice [39]. In a given experiment, all tenants have the same profile (naive, advanced or normal). We compare the results achieved by OFC to those of our baseline, OWK-Swift.

Results. Figure 9 reports the total execution time for all invocations of each function, per each tenant profile. For each scenario, OFC always outperforms OWK-Swift, with an improvement between 23.9 % and 79.8 % (54.6 % in average). For most functions, we observe slightly better results with *naive* tenants than with *advanced* ones (2–3 % of difference), because, in the *naive* case, OFC's memory capacity is larger (thus yielding more cache hits) than in *advanced* (see Figure 10).

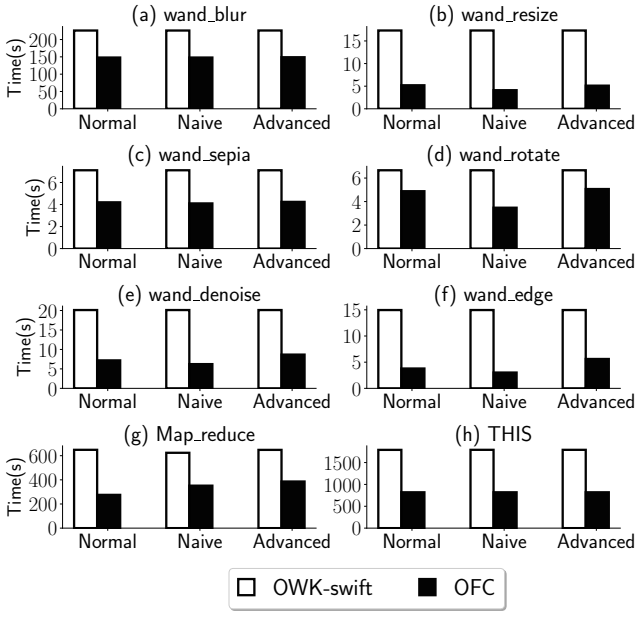

Figure 9. Sum of the execution times of all invocations for each function in three scenarios (distinct tenant profiles).

Table 2 reports internal OFC metrics during these experiments. First, we note that there was no abrupt function termination due to memory shortage (line 9). Second, the cache hit rate is high (up to 98.9 % for *naive*, line 10). However, there is a high rate of scale-up/down operations (line 1-5) due to the variability of function inputs (hence the need to reclaim and add memory from/to the cache). However, this does not really impact the overall function execution time since cache scaling up/down takes a negligible time

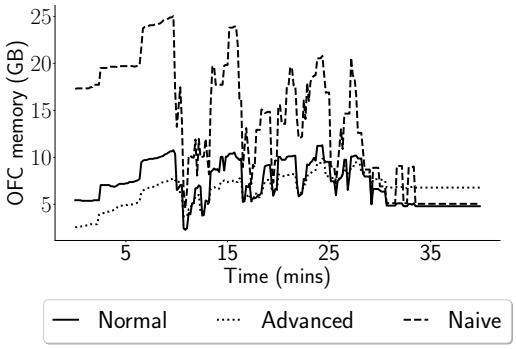

Figure 10. OFC's cache size evolution throughout the three experiments.

(line 6). For the above workload, the data is relatively easy to cache but, as noted previously, OFC provides benefits even on cache misses due to optimizing the load stage and caching intermediate objects during multi-stage requests. To show this, we also run the experiment with more tenants, 24 (3 per function) instead of 8 (1 per function). Due to space constraints, we present only a summary of the results. We observe a lower hit ratio of up to 32.3 %; yet, no failed invocation due to memory pressure is experienced (regardless of the tenant profiles). Besides, OFC's latency improvements fall from 23.9—79.8% to 4.5—44.9% due to the lower hit ratio.

Table 2. OFC internal metrics observed for the macro workloads with 8 tenants and three different user profiles.

	Metrics	Normal	Advanced	Naive
1	# Scale up	96	94	95
2	Total scale up time (s)	28.8	28.2	28.5
3	# Scale down (no eviction)	225	224	226
4	# Scale down (migration)	7	4	4
5	# Scale down (eviction)	0	0	0
6	Total scale down times (s)	85.4	81.2	83.2
7	# Bad predictions	7	7	7
8	# Good predictions	231	230	232
9	# failed invocations	0	0	0
10	Cache hit ratio (%)	98.21	93.12	98.9
11	Ephemeral data generated (GB)	300	300	300

8 Related Work

FaaS performance bottlenecks. We have previously discussed FaaS systems that use "serverful" (i.e., non-serverless) components as workarounds to mitigate performance bottlenecks stemming from shared/persistent sate management [9, 23, 32]. Below, we focus on other works.

Cloudburst [40], designed concurrently to OFC, also uses caches co-located with function executors and seamlessly supports existing functions. Cloudburst relies on the Anna key-value store, which introduces specific assumptions in terms of consistency semantics and protocols between the

FaaS workers and the backend storage service. Cloudburst leverages relaxed data consistency for maximum scalability and availability, whereas OFC is geared towards stronger consistency and persistence guarantees to support a broader set of use cases (e.g., hybrid FaaS/non-FaaS workloads interacting through a shared remote storage). Moreover, Cloudburst's authors do not discuss in details how the worker caches are provisioned and sized. OFC's memory hoarding/prediction techniques could be leveraged by Cloudburst.

Faasm [38] accelerates data movement between function instances, through the use of shared memory, both within a worker node and across worker nodes, using an abstraction akin to a distributed shared memory. Faasm relies on specific assumptions regarding the sandboxes runtime (language-based isolation) and the programming interface exposed to tenants for developing their applications.

Infinicache [43] leverages FaaS sandboxes and their keep-alive policy to implement an elastic in-memory caching service that is more cost-effective than traditional ones (e.g., Redis based services like AWS ElastiCache) for large objects. Infinicache relies on dedicated FaaS sandboxes (for the purpose of caching) that must be booked by Cloud tenants, who must also modify their applications to use the service.

OFC differs from the above works by offering full transparency for legacy cloud functions, no restriction on the choice of language or runtime for the functions, and only minor modifications of the FaaS and the backend storage infrastructure. Unlike the above systems, it harvests existing idle memory and does not require tenants nor cloud operators to provision and dimension dedicated resources for storage and data exchanges. Furthermore, OFC predicts memory usage and caching efficiency via ML techniques.

Lambada [28] is a specialized framework (for interactive data analytics) aimed at mitigating the performance of FaaS platforms, without any "serverful" component, thanks to domain-specific optimizations. Our work is focused on transparent and generic optimizations for FaaS applications based on the "ETL" pattern [13]. Boxer [46] has recently improved Lambada by enabling direct network communication (and hence, direct data exchange) between function instances, which could also bring benefits to a broader range of use cases [14]. OFC's approach remains useful even when direct communications between functions are possible, because it accelerates the "E" and "L" phases of the "ETL" pattern (very common in FaaS applications, not only in function pipelines).

FaaSCache [15] helps fine tuning the keep-alive policy of a FaaS platform by leveraging insights from the well-established caching literature. OFC is complementary to this approach, which does not address data caching and exchange, nor mitigation of memory waste caused by input variability.

Machine learning for resource management. A number of works have leveraged ML to optimize server applications and cloud infrastructures [2, 10, 11, 16, 18, 25]. We focus here on the most closely related to our work.

Resource Central [10] is used within Azure to collect telemetry data of resource usage in virtual machines (VMs), learn (offline) the behavior of these VMs, and provide a service for online predictions to resource managers (e.g., VM placement decisions). The authors mention examples based on different ML algorithms for the prediction of various metrics regarding the resource usage and lifetime of VMs. Our work considers the case of function invocations, which have very small durations and "white box" inputs.

COSE [2] uses statistical learning to determine the best configuration (w.r.t. SLAs and cost) for a cloud function. In contrast, our work aims at predicting the memory requirements and I/O-sensitivity of a function, in order to transparently mitigate storage performance bottlenecks, a major source of cost and performance overheads in FaaS workloads.

The Monitorless project [18] studied several ML approaches to infer the performance degradation of non-FaaS cloud applications, and opted for RandomForest despite long classification times. OFC requires fast classification since it uses the ML model on the critical path of function invocations.

Seer [16] uses deep learning and monitoring to infer the cause of QoS violations in microservices-based applications. For issues attributed to memory capacity, Seer resizes the resources of the corresponding container. Seer is not aimed at predicting memory consumption on a per-request basis.

9 Conclusion

We have introduced OFC and shown that such a caching layer allows significant performance improvements for the execution of diverse FaaS workloads in a cost-effective manner. Moreover, OFC's approach can be retrofitted in existing cloud infrastructures (FaaS platforms and object storage services) with limited modifications, is fully transparent for application-level code, and does not require to explicitly book or provision additional storage resources.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful feedback. Experiments presented in this paper were carried out using the Grid'5000 testbed[5]. This work has been partially funded by: Inria associated team MLNS2 with ENSP Yaoundé in Cameroon, the ANR Scalevisor ANR-18-CE25-0016 project, the "Studio virtuel" project of BPI and ERDF/FEDER (grant agreement number 16.010402.01), the "HYDDA" project of BPI Grant, the "IDEX IRS" (COMUE UGA grant), NSF CNS-1823236, and NSF CNS-1837382.

[5]Grid'5000 is supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

A Artifact Appendix

We make every component of OFC and its evaluation publicly available[6] under a form that was refined during the conference's Artifact Evaluation review process. We describe below all the parts that make OFC. The installation and usage guidelines are available in the repository.

At the core of OFC is a FaaS platform, Apache OpenWhisk, which we modified as described in the paper, mostly to add the ML module that learns models and uses their predictions. We integrated OpenStack Swift, a persistent object storage service, and a caching system, RAMCloud, both of which are bundled in OFC's repository. In addition, we developed a custom function runtime to provide the caching service to users' applications. The cache's memory pool is fed by the unused parts of the memory booked for the functions, which is predicted using machine learning. The repository also includes the infrastructure and data used to train the ML module offline. Finally, we include FaaSLoad: it is a load injector that we used to evaluate OFC, but it is also capable of monitoring function executions, which served to produce the training datasets for the ML work.

In the repository, folders are organized as follows:

- **customRuntime**: OWK function runtime image that embeds the proxy and write-back routines;
- **faasLoad**: load injector and dataset generator that can emulate many tenants with varied workloads;
- **functions**: OpenWhisk functions used as examples;
- **IMOC**: custom RAMCloud, OWK with ML module and Swift code bases as integrated into OFC;
- **machine-learning**: offline machine-learning scripts and data from the initial experiments.

[6] GitLab public repository: https://gitlab.com/lenapster/faascache. Note that OFC's name was changed, so references to its old name "FaaSCache" may linger.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In *Proceedings of the 2020 IEEE International Conference on Computer Communications (INFOCOM 2020)*.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[4] AWS News Blog. 2019. Provisioned Concurrency for Lambda Functions. https://aws.amazon.com/fr/blogs/aws/new-provisioned-concurrency-for-lambda-functions/. Accessed: 2021-02-16.

[5] AWS News Blog. 2020. Amazon S3 Update – Strong Read-After-Write Consistency. https://aws.amazon.com/fr/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/. Accessed: 2021-02-16.

[6] AWS News Blog. 2020. New for AWS Lambda – Functions with Up to 10 GB of Memory and 6 vCPUs. https://aws.amazon.com/fr/blogs/aws/new-for-aws-lambda-functions-with-up-to-10-gb-of-memory-and-6-vcpus/. Accessed: 2021-02-16.

[7] Leo Breiman. 2001. Random Forests. *Machine learning* 45, 1 (2001), 5–32.

[8] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[9] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy H. Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 13–24. https://doi.org/10.1145/3357223.3362711

[10] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[11] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[12] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS'20)*. ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[13] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. 2019. USETL: Unikernels for Serverless Extract Transform and Load Why Should You Settle for Less?. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hangzhou, China) *(APSys '19)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3343737.3343750

[14] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[15] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive With Greedy-Dual Caching. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM.

[16] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/

10.1145/3297858.3304004

[17] Google Cloud. [n.d.]. Google Cloud Storage Documentation — Consistency. https://cloud.google.com/storage/docs/consistency. Accessed: 2021-02-16.

[18] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. 2019. Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 149–162. https://doi.org/10.1145/3361525.3361543

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

[20] Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. 2005. Stress-testing Hoeffding Trees. In *European conference on principles of data mining and knowledge discovery*. Springer, 495–502.

[21] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[23] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[24] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 71–86. https://doi.org/10.1145/2815400.2815416

[25] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-Based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 541–556. https://doi.org/10.1145/3373376.3378525

[26] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[27] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[28] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 115–130. https://doi.org/10.1145/3318464.3389758

[29] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[30] Apache OpenWhisk. [n.d.]. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed: 2020-06-10.

[31] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, and et al. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (Aug. 2015), 55 pages. https://doi.org/10.1145/2806887

[32] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[33] David Durst Qian Li, James Hong. [n.d.]. Thousand Island Scanner (THIS): Scaling video analysis on AWS lambda. https://github.com/qianl15/this. Accessed: 2020-06-30.

[34] J Ross Quinlan. 1983. Learning Efficient Classification Procedures and their Application to Chess End Games. In *Machine learning*. Springer, 463–482.

[35] J Ross Quinlan. 1993. C4. 5: Programs for Machine Learning. (1993).

[36] Ran Ribensaft. [n.d.]. What AWS Lambda's Performance Stats Reveal. https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/. Accessed: 2020-02-03.

[37] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. 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. https://www.usenix.org/conference/atc20/presentation/shahrad

[38] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association. https://www.usenix.org/conference/atc20/presentation/shillaker

[39] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2019. Archipelago: A Scalable Low-Latency Serverless Platform. *CoRR* abs/1911.09849 (November 2019). arXiv:1911.09849 https://arxiv.org/abs/1911.09849

[40] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf

[41] OpenStack Swift. [n.d.]. OpenStack Swift. http://swift.openstack.org. Accessed: 2020-06-10.

[42] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. 2012. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 423–436. https://doi.org/10.1145/2150976.2151021

[43] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[44] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[45] Lavoisier Wapet, Alain Tchana, Giang Son Tran, and Daniel Hagimont. 2019. Preventing the propagation of a new kind of illegitimate apps. *Future Generation Computer Systems* 94 (2019), 368–380. https://doi.org/10.1016/j.future.2018.11.051

[46] Michal Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR 2021)*. https://doi.org/10.3929/ethz-b-000456492

[47] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery. https://doi.org/10.1145/3419111.3421280