Package-Aware Scheduling of FaaS Functions

Cristina L. Abad
Escuela Superior Politécnica del Litoral, ESPOL, Ecuador
cabad@fiec.espol.edu.ec

Edwin F. Boza
Escuela Superior Politécnica del Litoral, ESPOL, Ecuador
eboza@fiec.espol.edu.ec

Erwin van Eyk
TU Delft, The Netherlands
Platform9 Inc., USA
E.vanEyk@atlarge-research.com

ABSTRACT

We consider the problem of scheduling small cloud functions on serverless computing platforms. Fast deployment and execution of these functions is critical, for example, for microservices architectures. However, functions that require large packages or libraries are bloated and start slowly. A solution is to cache packages at the worker nodes instead of bundling them with the functions. However, existing FaaS schedulers are vanilla load balancers, agnostic of any packages that may have been cached in response to prior function executions, and cannot reap the benefits of package caching (other than by chance). To address this problem, we propose a package-aware scheduling algorithm that tries to assign functions that require the same package to the same worker node. Our algorithm increases the hit rate of the package cache and, as a result, reduces the latency of the cloud functions. At the same time, we consider the load sustained by the workers and actively seek to avoid imbalance beyond a configurable threshold. Our preliminary evaluation shows that, even with our limited exploration of the configuration space so-far, we can achieve 66% performance improvement at the cost of a (manageable) higher node imbalance.

CCS CONCEPTS

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

KEYWORDS

functions-as-a-service; scheduling; serverless computing; cloud computing; load balancing

ACM Reference Format:
Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. 2018. Package-Aware Scheduling of FaaS Functions. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3185768.3186294

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186294

1 INTRODUCTION

The *serverless computing* paradigm [8, 19] is increasingly being adopted by cloud tenants as it facilitates the development and composition of applications while relieving the tenant of the management of the software and hardware platform. Moreover, by making the server provisioning transparent to the tenant, this model makes it straightforward to deploy scalable applications in the cloud.

Within the context of serverless computing, the *Function-as-a-Service (FaaS)* model enables tenants to deploy and execute cloud functions on the cloud platform. *Cloud functions* are typically small, stateless, with a single functional responsibility, and are triggered by events. The FaaS cloud provider takes care of managing the infrastructure and other operational concerns, enabling developers to easily deploy, monitor, and invoke cloud functions [19]. These functions can be executed on any of a pool of servers managed by the cloud provider and potentially shared between the tenants.

The FaaS model holds good promise for future cloud applications, but raises new performance challenges that can hinder its adoption [18]. One of these performance challenges is the scheduling or mapping of cloud functions to a specific worker node, as this task may entail conflicting goals [8, 13, 18]: (1) Minimize node imbalance, (2) maximize code locality, and (3) maximize data locality. Current load balancers already achieve (1), while (3) is only a goal of data-intensive workflows (and as such, the workflow scheduler should work in conjunction with the function scheduler to achieve this goal). In this work, we focus in achieving (2), which is becoming progressively more important as the number, complexity, and desired performance requirements of cloud functions increases.

Small cloud functions can be launched rapidly, as they run in preallocated virtual machines (VMs) and containers. However, when these functions depend on large packages their launch time slows down; this affects the elasticity of the application, as it reduces its ability to rapidly respond to sharp load bursts [13][1]. Moreover, long function launch times have a direct negative impact on the performance of serverless applications using the FaaS model [18]. A solution is to cache packages at the worker nodes, leading to speed-ups of up to 2000x when the packages are preloaded prior to function execution instead of having to bundle them with the cloud function (for workloads that require only one package) [14]. In sum, code locality improves performance as it reduces the time that it takes to load packages, and thus, reduces request latency.

Existing FaaS schedulers—like those from OpenWhisk, Fission and OpenLambda—are simple load balancers, unaware of any packages that may have been cached and preloaded in response to prior

[1] For simplicity, in this paper we talk about *large* packages, but the start-up time is not only due to having to download the package; the local installation and run-time import processes also add overhead. The whole process can take on average more than four seconds, with close to half of that time attributable to the download time [13, 14].

function executions, and therefore cannot reap the benefits of package caching (except by chance). To address this problem, we propose a novel approach to scheduling cloud functions on FaaS platforms with support for caching packages required by the functions. Towards this end, our contributions consist of the following:

(1) We present the preliminary design of our package-aware scheduler for FaaS platforms in section 3. The proposed algorithm aims to maintain a good balance between maximizing cache affinity and minimizing the node imbalance.
(2) Besides the proposed algorithm for pull-based scheduling, we identify potential extensions for alternative forms of scheduling: push-based scheduling in section 3.4, distributed scheduling in section 3.5, and how to extend the algorithm to deal with multiple packages in section 3.6.
(3) In section 4 we present a preliminary evaluation of our package-aware scheduling algorithm. Using simulation with synthetic workloads we demonstrate that our approach can improve function latency at the cost of node imbalance.

2 BACKGROUND: PACKAGE CACHING WITH PIPSQUEAK

Our scheduling algorithm assumes that there is a package cache at each worker node within the FaaS platform. In particular, we plan on using the Pipsqueak package cache for the OpenLambda open-source FaaS platform [8, 13]. The goal of Pipsqueak is to reduce the start-up time of cloud functions via supporting lean functions whose required packages are cached at the worker nodes.

Pipsqueak maintains a set of Python interpreters with packages pre-imported, in a sleeping state. When a cloud function is assigned to a worker node, it checks if the required packages are cached. To use a cached entry, Pipsqueak: (1) Wakes up and forks the corresponding sleeping Python interpreter from the cache, (2) relocates its child process into the handler container, and (3) handles the request. If a cloud function requires two packages that are cached in different sleeping interpreters, then only one can be used and the missing package must be loaded into the child of that container (created by step 2 above). To deal with cloud functions with multiple package dependencies, Pipsqueak supports a tree cache in which one entry can cache package A, another entry can cache package B, and a child of either of these entries can cache both packages.

Having pre-initialized packages in sleeping containers speeds up function start-up time because it eliminates the following steps present in an unoptimized implementation: (1) downloading, (2) installing, and (3) importing the package. The last step also includes the time to initialize the module and its dependencies. Especially for cloud functions with large libraries, this process can be extremely time consuming, as it can take 4.007s on average [14].

3 PROPOSED DESIGN

In this section, we describe the goals and preliminary design of our function scheduler. We use the generic terms *task* and *worker* to describe the design. Tasks are cloud functions that need to be executed on worker nodes. A worker node is capable of running many tasks simultaneously and can be, for example, a virtual machine managed by a container orchestration system such as Kubernetes.

3.1 System model and assumptions

In section 3.2 we outline the goals for a package-aware scheduler for FaaS functions; our proposed scheduling algorithm is described in section 3.3. This algorithm assumes a pull-based model where a centralized scheduler assigns tasks to queues. When a worker has spare capacity, it contacts the scheduler to get a function assignment from one of the functions at the head of the task queues. In other words, as shown in Figure 1, we assume a *centralized* scheduler from which tasks are *pulled* by the worker nodes. We discuss how to relax these assumptions for push-based scheduling in section 3.4 and for distributed scheduling in section 3.5.

Package caching. We assume that there is a package or library caching mechanism implemented at the worker level; as described in section 2. In this work, we consider the case where the sleeping containers in the package cache have been preloaded with single packages. This means, that if a cache has preloaded packages A and B, it would have done so in independent sleeping containers, and a function requiring both A and B can only leverage one of the two. In case of a function depending on more than one package, the additional packages would need to be loaded on-demand. For more details on how the Pipsqueak package cache works, see section 2.

Our scheduler is agnostic of the contents of the worker caches. An alternative approach would be to keep track of the information of which packages are cached by which workers. However, we did not pursue this idea as we suspect that this approach would impose a significant overhead on the system (network communications, resources to store, and managing the caching directory component).

We seek to achieve scheduling affinity for the largest package required by a task, as this is the package that is most useful to accelerate its loading time. In section 3.6 we discuss how to extend the algorithm to consider multiple package requirements.

3.2 Conflicting goals

To **balance the load**, a single first-come-first-served (FCFS) queue is sufficient for the pull-based model. The analogous approach in the push-based model is to use a Round-Robin assignment, though this is not optimal, as the resource consumption of tasks may vary significantly [11]. Better alternatives are Join-the-Shortest-Queue (JSQ) [7] and Join-Idle-Queue (JIQ) [11].

To **maximize cache affinity** (of the package cache), we can use consistent hashing [9] to assign all tasks that require a particular package to the same worker.[2] However, as package popularity is not uniformly distributed, this approach would create hot-spots, overloading workers that cache popular packages.

In this paper, *our goal is to maintain a good balance between maximizing cache affinity and minimizing the node imbalance.*

3.3 Proposed scheduling algorithm

Algorithm 1 shows the details of the proposed procedure. The scheduler keeps track of one FIFO scheduling queue per worker, and uses hashing to try to assign all tasks that require the same package to the same worker, to encourage cache affinity. To avoid overloading a worker to which one or more popular packages map, a configurable

[2] This is in case we are optimizing only for affinity with the largest package required by each task. To maximize affinity of multiple packages simultaneously, we could model this as a mathematical optimization problem.

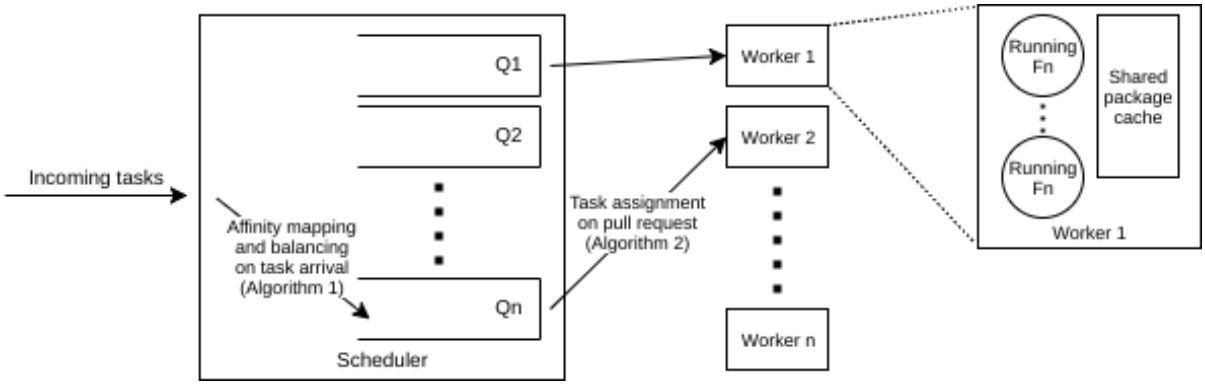

Figure 1: Model assumed in the algorithm proposed in section 3.3. The scheduler assigns incoming tasks to queues, based on package affinity while avoiding node imbalance (design goals). The worker nodes have a shared package cache that can be leveraged by the cloud functions to speed up the startup times. Variations of the algorithm suitable for a push-based scheduling, distributed scheduling and multi-package affinity are discussed in sections 3.4, 3.5, and 3.6, respectively.

Algorithm 1: Queue assignment algorithm (scheduler)

Global data: List of workers, $W = w_1, ..., w_n$, list of scheduling queues $Q = q_1, ..., q_n$, such that q_i corresponds to the functions assigned to w_i, Hash functions H_1 and H_2, maximum load threshold, t

Input: Function id, f_{id}, largest package required by task, p_l

1 **if** (pl *is not NULL*)**then**
  `/* Calculate two possible worker targets */`
2 $t1 = H_1(p_l)\%|W| + 1$
3 $t2 = H_2(p_l)\%|W| + 1$
  `/* Select target with least load */`
4 **if** ($length(q_{t1}) < length(q_{t2})$)**then**
5 $A := t1$
6 **else**
7 $A := t2$
  `/* If target is not overloaded, we are done */`
8 **if** ($length(q_A) < t$)**then**
9 Insert f_{id} into q_A
10 **return**
  `/* Try to balance load */`
11 Insert f_{id} into shortest queue, q_i

Algorithm 2: Task-worker mapping algorithm (scheduler)

1 `FnGetTaskAssignment(w) /* called by worker w */`
2 **if** (q_w *is not empty*)**then**
3 **return** Front task from q_w
4 **else**
  `/* work stealing step */`
5 **return** Front task from longest queue

maximum load threshold is used. If the scheduler cannot achieve affinity without assigning a task to an overloaded node (defined as one for which its task queue has exceeded the threshold, t), then the scheduler chooses the shortest worker queue. To improve cache affinity while improving load balance, we apply the power-of-2 choices technique [12], by using two hashing functions to map a task to a queue; each hash function maps the task to a different queue, and the task is assigned to the shortest of those queues.

When a worker has spare capacity, it contacts the scheduler to request a task assignment (Algorithm 2). The scheduler assigns the task to the worker at the front of the queue corresponding to that worker. If the queue is empty, the scheduler selects the front task from the longest queue, which is known as the *work stealing* step.

3.4 Push-based model

Schedulers can use pull or push-based models, as described next. In the *pull*-based approach, the scheduler assigns tasks to one or more queues. When a worker has spare capacity, it contacts the scheduler and gets assigned the front task from one of the queues. The scheduling algorithm determines how the tasks are placed on the queues, and from which queue a worker pulls a task. The tasks in these queues are serviced in FIFO order. With the *push*-based approach, upon task arrival, the scheduler maps a task to a worker and sends the task to the worker, which will either execute it immediately using a processor sharing approach, or will queue it locally until it has spare capacity. Examples of frameworks that use the pull-based approach to scheduling are OpenWhisk and Kubeless; Fission and OpenLambda instead use a push-based approach.

The algorithms proposed in section 3.3 cannot be directly ported to a push-based scheduler for two reasons: (1) When selecting the least-loaded worker, the scheduler cannot exactly know the length of the task queues at each worker, and (2) in the work stealing step, a worker cannot know which of the other queues is the longest.

To deal with issue (1), the scheduler can keep track of the load being sustained by each worker (requests per second); however, this fails if the size of the tasks is unbalanced, and some workers could become overloaded. Alternatively, workers could periodically report their load (queue length) to the scheduler, as in JSQ [7].

Regarding issue (2), to avoid having to communicate the load between the workers, there are two possible solutions: (a) Have the worker ask the scheduler which worker to steal work from, or (b) use the power-of-2 choices approach and have a worker poll two other random workers and steal a task from the most loaded one.

3.5 Distributed scheduler

If the load of the FaaS platform is large enough that scheduling decisions cannot be made in a reasonably short time, then the scheduling load can be distributed between a set of scheduler nodes [2]. We call this, a *distributed scheduler*.

In the case of a distributed scheduler, it is not a good idea to try to have all the scheduler nodes share perfect knowledge of the length of the queues [11]. Furthermore, if each scheduler decides a mapping from the task to worker independently, this could result in overloading the worker that had the shortest queue. The alternative would be for the schedulers to use a consensus algorithm, although this would add additional overhead on the critical path.

To avoid this problem in a distributed scheduling scenario, we propose changing the Join-the-Shortest-Queue component of our scheduler with the Join-Idle-Queue algorithm [11], which decouples discovery of lightly loaded servers from job assignment, thus leading to very fast task assignments.

3.6 Affinity with multiple required packages

The simplest way to extend our algorithm for the multiple package case is to use a greedy approach in which we only try to achieve affinity for the largest package. If the nodes possibly caching the package are overloaded, then we try to achieve affinity with the next largest package, and so on.

Alternatively, we could map a task to a worker that has affinity to multiple packages the function needs (modeling this as a mathematical optimization problem). However, our current solution assumes we cannot leverage multiple cached packages, as they would be preloaded in different sleeping interpreters (see sections 2 and 3.1); we leave relaxing this assumption for future work.

3.7 Caching policy

Our algorithm is agnostic to the caching policy being used at the workers. However, to maximize the effectiveness of our approach we can co-design a caching policy that takes advantage of the knowledge of which packages are affinity packages for the current worker. Towards this goal, we propose to divide the memory into two caching segments: S_1, which will hold the affinity packages, and S_2, which can cache any type of package. The reasoning is that it may be useful to cache very popular packages, even if they are not considered affinity packages for the node. Algorithm 3 describes how we decide whether to cache a package or not.

For the evaluations in section 4, we only consider the extreme cases when the size of S_1 is 0, and when the size of S_2 is 0. In other words, we only evaluated the use of only a regular (LRU) cache, and the alternative of only caching affinity packages. In future work, we will assess how the segmenting of the cache affects the overall hit rate, and we will consider alternative policies to LRU.

Algorithm 3: Caching policy (called upon a cache miss)

Global data: Hash functions H_1 and H_2, Cache segments, S_1 and S_2, Number of workers, n, Current worker id, w

Input: Package, p

```
/* Calculate affinity workers for package        */
```

1 $t1 = H_1(p)\%n + 1$

2 $t2 = H_2(p)\%n + 1$

```
/* Does current worker have affinity for p?      */
```

3 **if** ($w == t1$ *or* $w == t2$)**then**

4 Cache p in S_1

5 **else**

6 Cache p in S_2

4 PRELIMINARY EVALUATION

In this section, we present preliminary results of a simulation-based evaluation of our algorithm. We implemented the simulator in Python, using the SimPy simulation framework[3] and ran tests using the following **configuration parameters**:

- Arrivals are exponentially distributed, with a mean inter-arrival time of $0.1ms$.
- The number of worker nodes is 1 000; each worker can run as many as 100 tasks simultaneously ($st = 100$).
- The popularity of the packages is given by a Zipf distribution, with parameter $s = 1.1$.
- The time to start the packages—including time to download, install and import—is randomly sampled from an exponential distribution with an average time to start of $4.007s$.
- Each function requires a random number of packages, sampled according to an exponential distribution, with an average number of 3 required packages.
- Each worker has a LRU package cache (capacity = $500MB$).
- The sizes of the (cacheable) packages is modeled after the sizes of the packages in the PyPi repository.
- Time to launch a function that requires no packages: $1s$.
- The running time of a task (after loading required packages) is exponentially distributed with mean = $100ms$.
- Experiment duration: 30 minutes.
- Overload threshold: $t = st = 100$.
- In all cases, the eviction policy is LRU.

While the configuration described above represents an artificial scenario, the configuration values were chosen to closely model real observed behavior, as reported by related work [8, 10, 13]. In the future, we plan to expand our evaluation to include trace-based evaluations, as well as experiments in a public cloud.

We implemented four **scheduling policies** and compare their performance regarding: (1) How well they balance the load, (2) the package cache hit rate, and (3) the latency of each cloud function (task time in system). The scheduling policies we implemented are:

(1) Join-the-Shortest-Queue (JSQ_{rc}): Scheduler keeps one task queue for each worker. A new task is added to the shortest

[3] https://pythonhosted.org/SimPy/

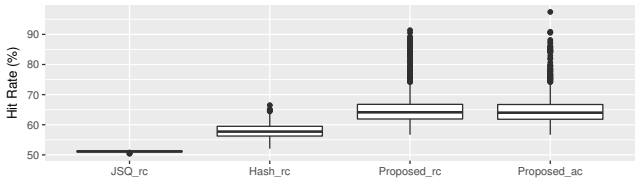

Figure 2: Box plots of the hit rates of the package caches at the worker nodes. Our algorithm improves the average hit rate by actively seeking to improve package-affinity.

Table 1: Task latency percentiles (in seconds). Our algorithm improves latency due to improved cache hit rate.

Algorithm	$50th$	$90th$	$95th$	$99th$
$\mathcal{JSQ}_{rc}$	3.95	18.53	25.29	40.86
$Hash_{rc}$	4.43	277.81	606.12	1196.47
$Proposed_{rc}$	1.36	11.03	16.21	28.88
$Proposed_{ac}$	1.36	11.00	16.11	29.42

queue. Queues are served in FIFO order. We evaluated this policy with a per-worker package cache.

(2) Hash-based cache affinity ($Hash_{rc}$): A hash function applied to the largest package required by the cloud function determines the mapping of a function to a worker. A per-worker package cache was used.

(3) $Proposed_{rc}$: Our proposed algorithm, with the greedy approach to seeking affinity when multiple packages are required (section 3.6), and a per-worker package cache.

(4) $Proposed_{ac}$: Similar to $Proposed_{rc}$, but with a different caching policy: per-worker package cache that caches *only* those packages that have affinity to it (as determined by the functions H_1 and H_2 in Algorithm 1; see section 3.7).

Our preliminary **results** show we can improve the median **hit rate** from 51.2% ($\mathcal{JSQ}_{rc}$) to 64.1% ($Proposed_{rc}$), as shown in Figure 2. The proposal to cache only affinity packages ($Proposed_{ac}$) produced results very similar to those of $Proposed_{rc}$. In the future, we plan on evaluating a split cache, as described in section 3.7, to see if there is value in reserving some cache space for affinity packages, both for Zipfian and real workloads.

The improved hit rate has a positive effect on the **latency** of the tasks, as shown in Table 1. Median latency improves by 65.6% ($Proposed_{rc}$ vs. $\mathcal{JSQ}_{rc}$), and tail latency improves by 40.5% ($90th$ percentile). Note that we avoid the straw man fallacy of comparing our algorithm against $\mathcal{JSQ}$ with no caching, as this is an unfair comparison. Both $\mathcal{JSQ}_{rc}$ and $Proposed_{rc}$ are much better than $\mathcal{JSQ}$ with no caching; the former improves median latency by 65 times, while our algorithm improves median latency by 189 times.

Finally, we can quantify how well each scheduling algorithm **balances the load** using the *coefficient of variation*, which is a measure of dispersion defined as the ratio of the standard deviation to the mean: $c_v = \sigma/\mu$. We count the total number of tasks assigned to each worker, and report the coefficient of variation in Table 2[4].

[4]This simple metric quantifies the dispersion in the total work assigned to each worker; in the future, we will study how the load changes during the test duration.

Table 2: Node unbalance, measured using the coefficient of variation of the number of functions assigned to each worker (smaller is better).

Scheduling Algorithm	c_v
$\mathcal{JSQ}_{rc}$	1.02
$Hash_{rc}$	357.65
$Proposed_{rc}$	65.33
$Proposed_{ac}$	66.06

We can observe that our algorithm sacrifices some unbalance, to seek a higher hit rate (and smaller latency). $\mathcal{JSQ}$ achieves near perfect balancing, while the hash-based affinity algorithm produces the most unbalanced task assignments.

Discussion. The main reason for load balancing is to improve performance, as tasks that are assigned to overloaded workers are bound to be delayed in their completion. However, the moderate unbalance of our proposed algorithm is not necessarily an issue, as our experiments show that we actually improve performance: tasks that run on workers that have preloaded a required package, take significantly less time to finish; thus, by improving the cache hit rate, we improve overall system performance. This is not the case for the $Hash_{rc}$ algorithm, for which the worker overload is too high, taking a significant toll on performance (tasks are 15 and 29 times slower for the $90th$ and $99th$ percentiles, when compared to JSQ_{rc}). Although the initial results are promising, more experimentation should be done to better understand the limitations of our approach. This submission seeks early feedback on our proposal, as well as encouraging discussion from the Cloud Performance community about future directions in improving FaaS performance.

5 RELATED WORK

We build upon a large body of work in task scheduling. Early work in affinity scheduling sought to improve performance in multiprocessor systems by reducing cache misses via preferential scheduling of a process on a CPU where it has recently run [5, 17]. However, the issue here is not how to map threads to CPUs, but how to re-schedule them soon enough to reap caching benefits, while at the same time avoiding unfairness and respecting thread priority.

Better related to the problem studied in this paper, is the case of locality- or content-aware request distribution in Web-server farms [3]. In this context, the simplest solution is static partitioning of server files using URL hashing, to improve cache hit rate; though this could lead to extremely unbalanced servers. Others have proposed algorithms that partition Web content to improve cache hit rate, while monitoring server load to reduce node unbalance [3, 15]. While these solutions share some similarities with ours, they only try to improve the locality of the access to one Web object, as each HTTP request targets one object only. We consider the case of tasks that could require multiple objects (packages). We also differ in that we propose co-designing the worker caches (eviction policy) with the scheduler. Furthermore, the work in the Web domain typically assumed that the workloads are relatively stable, as was the case with traditional Web hosting systems. Modern cloud workloads are

significantly more dynamic, making solutions that require offline workload analysis (e.g., see [4]), inadequate for this domain.

We also build upon prior work in load balancing for server clusters. In this domain, there is work in centralized load balancing [7] and distributed load balancing [12]. For example, the Join-Idle-Queue (JIQ) algorithm incurs no communication overhead between the dispatchers and processors at job arrivals [11]; it does this by decoupling the discovery of lightly-loaded servers from job assignment, thus removing the load balancing work from the critical path of the request processing. This technique can be used to port our algorithm to a distributed scheduler (see section 3.5).

The near-data scheduling problem is a special case of affinity scheduling applicable to data-intensive computing frameworks like Hadoop, where each type of task has different processing rates on different subsets of servers [21]; tasks that process data that is stored locally execute the fastest, followed by tasks whose input data is stored in the same rack, followed by tasks whose input data is stored remotely (in a different rack). Several near-data schedulers have been proposed for Hadoop [20, 22, 23]. However, these are not directly applicable to the problem studied in this paper, as they require a centralized directory to keep track of the location of the data blocks (i.e., the namenode in Hadoop). In contrast, our proposed algorithm uses hash-based affinity mapping, a mechanism that has a minimal overhead and requires no centralized directory. Implementing a directory of cached packages in a serverless computing platform would impose significant overhead on the infrastructure, as extra communication and storage would be required. Moreover, unlike data block storage in Hadoop, the contents of a package cache could change rapidly, as packages can enter and leave the cache frequently, leading to problems where the scheduler would assign tasks based on stale knowledge about the status of the caches.

Finally, our work joins recent efforts by other researchers in seeking to advance the state-of-the-art in the management of resources in serverless computing clouds and the containerized platforms that support them [6, 13, 16]. In particular, we were inspired by the recent work in caching packages in OpenLambda by Oakes et al. [13], though they left the global scheduling work (to improve cache hit rates) for future work.

6 CONCLUSIONS AND FUTURE WORK

Current scheduling approaches in Function-as-a-Service (FaaS) platforms are relatively simplistic, lacking awareness of cached packages required by cloud functions which could improve performance. Towards solving this problem, we propose a package-aware scheduling algorithm that attempts to optimize the use of cached packages versus maintaining a balanced load over the worker nodes. Our initial evaluation, based on simulation, shows that the latency of cloud functions can be reduced by up to 66% by our proposed algorithm at the expense of a higher node imbalance. These preliminary results encourage us to continue our research in this direction.

In the future we will implement the algorithm in OpenLambda and perform experiments in a public cloud, and perform more extensive simulations to answer interesting questions that arose during our study: Can we improve the cache hit-rates by giving caching preference to affinity packages? How do the results change when the skew of the popularity of the packages changes? What happens with workloads with significant temporal locality?[5] What is the optimal overload threshold value for a workload and how can we automatically tune this parameter? Would it be beneficial to use more than two affinity nodes for very popular packages? Is it a good idea to try to improve package-locality for Big Data functions that would possibly benefit more from seeking data-locality instead?

REFERENCES

[1] Abad, C., Yuan, M., Cai, C., Lu, Y., Roberts, N., and Campbell, R. Generating request streams on big data using clustered renewal processes. *Performance Evaluation 70*, 10 (2013).

[2] Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Marchal, L., and Robert, Y. Centralized versus distributed schedulers for bag-of-tasks applications. *IEEE Transactions on Parallel and Distributed Systems 19*, 5 (2008).

[3] Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. The state of the art in locally distributed Web-server systems. *ACM Comput. Surv. 34*, 2 (2002).

[4] Cherkasova, L., and Ponnekanti, S. Optimizing a content-aware load balancing strategy for shared web hosting service. In *Intl. Symp. Model., Anal. and Sim. of Comp. and Telecomm. Sys. (MASCOTS)* (2000).

[5] Feitelson, D. Job scheduling in multiprogrammed parallel systems. Tech. rep., 1997. IBM Research Report 19790.

[6] Ferreira, J., Cello, M., and Iglesias, J. More sharing, more benefits? A study of library sharing in container-based infrastructures. In *Intl. Conf. Par. Distrib. Comp. (Euro-Par)* (2017).

[7] Gupta, V., Harchol Balter, M., Sigman, K., and Whitt, W. Analysis of Join-the-Shortest-Queue Routing for Web server farms. *Perform. Eval. 64* (2007).

[8] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Serverless computation with OpenLambda. In *USENIX Work. Hot Topics in Cloud Comp. (HotCloud)* (2016).

[9] Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. Web caching with consistent hashing. *Comp. Netw. 31*, 11 (1999).

[10] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S. Serverless computing: An investigation of factors influencing microservice performance. In *IEEE Intl. Conf. Cloud Eng. (ICPE), to appear* (2018).

[11] Lu, Y., Xie, Q., Kliot, G., Geller, A., Larus, J., and Greenberg, A. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable Web services. *Perform. Eval. 68*, 11 (2011).

[12] Mitzenmacher, M. The power of two choices in randomized load balancing. *IEEE Trans. Par. Distrib. Sys. 12*, 10 (2001).

[13] Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Pipsqueak: Lean Lambdas with large libraries. In *IEEE Intl. Conf. Distrib. Comp. Sys. Workshops (ICDCSW)* (2017).

[14] Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Pipsqueak: Lean Lambdas with large libraries, 2017. (Presentation given at the) Workshop on Serverless Computing (WoSC).

[15] Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. Locality-aware request distribution in cluster-based network servers. *SIGOPS Oper. Syst. Rev. 32*, 5 (1998).

[16] Sampé, J., Sánchez-Artigas, M., García-López, P., and París, G. Data-driven serverless functions for object storage. In *ACM/IFIP/USENIX Middleware* (2017).

[17] Torrellas, J., Tucker, A., and Gupta, A. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing 24*, 2 (1995).

[18] van Eyk, E., Iosup, A., Abad, C. L., Grohmann, J., and Eismann, S. A SPEC RG cloud group's vision on the performance challenges of FaaS cloud architectures. In *(Under review)* (2018).

[19] van Eyk, E., Iosup, A., Seif, S., and Thömmes, M. The SPEC cloud group's research vision on FaaS and serverless architectures. In *Intl. Workshop on Serverless Comp. (WoSC)* (2017).

[20] Wang, W., Zhu, K., Ying, L., Tan, J., and Zhang, L. MapTask scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality. *IEEE/ACM Trans. Netw. 24*, 1 (2016).

[21] Xie, Q., and Lu, Y. Priority algorithm for near-data scheduling: Throughput and heavy-traffic optimality. In *IEEE Conf. Comp. Comm. (INFOCOM)* (2015).

[22] Xie, Q., Pundir, M., Lu, Y., Abad, and Campbell. Pandas: Robust locality-aware scheduling with stochastic delay optimality. *IEEE/ACM Trans. Netw. 25*, 2 (2017).

[23] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *European Conf. Comp. Sys. (EuroSys)* (2010).

[5] The *Independence Reference Model (IRM)*, used when sampling from Zipf, assumes no temporal locality [1].