TOWARDS IMPROVED TASK SCHEDULING IN CLOUD COMPUTING PLATFORMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF ESPOL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Edwin Boza
March 2022

© Copyright by Edwin Boza 2022
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Cristina Abad) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Federico Dominguez)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Luis Mendoza)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Julio Arauz)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Javier Bustos)

Abstract

With the microservices architectural model, complex systems are built as a set of small loosely-coupled independent components. Deployment of microservice-based applications takes advantage of cloud computing platforms to provide scalability, high availability, and to simplify the deployment and operation of applications. Improving the performance of microservices running on containerized and serverless cloud computing platforms is important for many applications with requirements like real-time, low latency, responsive auto-scaling and to increase user engagement.

This thesis aims to improve the performance of these applications through smart task scheduling and request routing decisions. We started our work enhancing function latency on serverless platforms by increasing code locality, and assessing conflicting goals for the scheduling process. We then studied the benefits of performance-aware scheduling decisions for containerized platforms, to improve microservices performance. We demonstrated that our affinity scheduling approach has important applicability beyond the microservices domain by increasing the use of the cache layer to reduce the access time to the data stored in a data lake. Finally, we also addressed the use of serverless-based microservices to support the dynamic implementation of self-adaptive cloud services.

Acknowledgments

I would like to thank to all the people who made possible to bring a conclusion to this work.

Contents

Abstract **iv**

Acknowledgments **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Research goals 2

1.3 Research questions 2

1.4 Main contributions 3

1.4.1 Fast deployment and execution of cloud functions in Function-as-a-Service (FaaS) platforms 3

1.4.2 A closed-loop system to maintain microservices performance in containerized cloud platforms 3

1.4.3 Smart scheduling beyond serverless-based microservices 4

1.4.4 FaaS-based architecture to optimize cloud component settings 4

1.5 Regional impact and technological transfer 4

1.6 Dissertation outline 5

2 Background **6**

2.1 Cloud computing 6

2.1.1 Serverless computing (FaaS) 6

2.1.2 Containers 9

2.2 Microservices 11

2.2.1 Definition of microservices 11

2.2.2 Platforms for the deployment of microservices 11

2.2.3 Challenges for the deployment of microservices in the cloud 11

3 Package-aware scheduling of FaaS functions **13**

3.1 Introduction 13

3.2 Proposed design 15
3.2.1 System model and assumptions 15
3.2.2 Conflicting goals 16
3.2.3 Proposed scheduling algorithm 16
3.2.4 Push-based model 17
3.2.5 Distributed scheduler 18
3.2.6 Affinity with multiple required packages 18
3.2.7 Caching policy 19
3.3 Preliminary evaluation 19
3.4 Related work 22
3.5 Conclusions 23

4 Package-aware task scheduling in real serverless platforms 24
4.1 Introduction 25
4.2 Design and implementation 25
4.2.1 Package-aware scheduling in olscheduler 26
4.2.2 Validation results 27
4.3 Related work 28
4.4 Conclusions 29

5 Package-aware task scheduling versus traditional load balancing algorithms 30
5.1 Introduction 31
5.2 Background and related work 33
5.2.1 FaaS and OpenLambda 33
5.2.2 Load balancing and scheduling 35
5.3 Problem definition 37
5.3.1 Assumptions and limitations 37
5.3.2 System model 38
5.3.3 Naive solutions 39
5.3.4 Recent applicable solutions 39
5.4 Proposed solution 40
5.4.1 Analysis 41
5.4.2 Implementation in OpenLambda 42
5.5 Experimental evaluation 43
5.5.1 Experimental setup 43
5.5.2 Experimental results 45
5.6 Conclusions 49

6 Performance-aware deployment of containers **50**
6.1 Introduction 50
6.2 Background and motivation 52
6.3 Design 53
6.4 Evaluation 55
6.4.1 Response times: Experimental design and results 56
6.4.2 Initialization times: Experimental design and results 59
6.5 Related work 60
6.6 Closing discussion 61

7 Cache affinity request routing with load control (CARLOs) for data lakes **63**
7.1 Introduction 63
7.2 Architecture 64
7.3 Proposed algorithm 65
7.4 Experimental design 66
7.5 Results 67
7.6 Conclusions 72

8 Serverless-based microservices to support self-adaptive cloud services **73**
8.1 Introduction 73
8.1.1 Contributions and chapter roadmap 74
8.1.2 Threats to validity 75
8.2 Background, motivation and related work 76
8.2.1 How important is (optimal) memory partitioning in cloud caches? 77
8.2.2 Architectures for self-partitioning cloud caches 78
8.2.3 Why aren't current cloud caches already self-partitioning? 80
8.2.4 Other related work 81
8.3 The memory partitioning problem 82
8.3.1 Solving the optimization problem 84
8.4 Design and implementation of SPREDS 86
8.4.1 Summary of how SPREDS works and outline for the remainder of the Section 88
8.4.2 Workload monitor 88
8.4.3 Optimization (solver) 89
8.4.4 Adaptation plan and execution 90
8.5 Experimental validation and cost analysis 90
8.5.1 Workloads 91
8.5.2 Performance overhead 92
8.5.3 Usefulness 93

8.5.4 Costs . . . 94
8.6 Open challenges . . . 95
8.7 Concluding remarks . . . 96

9 Conclusions and future directions **97**
9.1 Conclusions . . . 97
9.2 Future directions . . . 98

Bibliography **100**

List of Tables

3.1 Task latency percentiles for scheduling algorithms 21
3.2 Node imbalance, when using different scheduling algorithms 22

4.1 Task latency percentiles, when using different scheduling algorithms 28

5.1 Top ten required packages by Python and Java files in GitHub 38

6.1 Summary of the experiments for container-based environments 55

7.1 Throughput CARLOs vs LeastConnections 69
7.2 Throughput CARLOs vs Cache Affinity 70
7.3 ANOVA Evaluation 72

8.1 Summary of the recent work in the domain of self-partitioning cloud caches 78
8.2 Recent approaches for architecture of the self-partitioning caches 79
8.3 Parameters used in the model definition. 82
8.4 Amazon Web Services Products used in the SPREDS implementation. 88
8.5 Description of the testbeds used in experiments 91
8.6 Performance improvement of SPREDS versus Redis 94
8.7 AWS costs, for items enumerated in Figure 8.2 94

List of Figures

2.1 The OpenLambda architecture 8
2.2 The Pipsqueak package cache in the OpenLambda stack 8

3.1 Scheduling model for a serverless platform 15
3.2 Hit rate improvement with a package-aware routing algorithm 21

5.1 Example of function execution requests arriving at a FaaS scheduler 32
5.2 The OpenLambda architecture 33
5.3 Average hit rate improvement, when using a package-aware scheduling algorithm . . 45
5.4 Node imbalance, when using different scheduling algorithms 46
5.5 Per-task speedup, when using a package-aware scheduling algorithm 47
5.6 Task turnaround times, when using different scheduling algorithms 48
5.7 Threshold effect in latency, when using a package-aware scheduling algorithm 48

6.1 Proposed design of a performance-aware system for container deployment 54
6.2 Average response times for a microservice, when using different placement algorithms 56
6.3 Median response times for a microservice, when using different placement algorithms 57
6.4 Response times in transition phase for a microservice, when using different placement algorithms 58
6.5 Response times in steady phase for a microservice, when using different placement algorithms 58
6.6 Initialization times for a microservice, under CPU stress 59
6.7 Performance degradation of container initialization times 60

7.1 Proposed architecture for a CEPH-based data lake 65
7.2 Latency Boxplot CARLOs vs LeastConnections 68
7.3 Time Series Average Latency CARLOs vs LeastConnections 69
7.4 Time Series Throughput CARLOs vs LeastConnections 70
7.5 Latency Boxplot CARLOs vs Other Cache affinity algorithms 71

8.1 Motivating architecture for self-adaptive cloud services 76
8.2 Components of SPREDS and their interactions 87
8.3 Average running time of optimization algorithms 90
8.4 Request latency for Redis and SPREDS 92
8.5 Cache partition sizes differences with SPREDS 93
8.6 Costs of running the autonomic controller on AWS 95

List of Algorithms

1 Queue assignment algorithm (scheduler) 17
2 Task-worker mapping algorithm (scheduler) 17
3 Caching policy (called upon a cache miss) 19

4 Package-aware scheduler algorithm for OpenLambda 27

5 Package-aware scheduler algorithm (PASch) 41
6 Mapping function for PASch algorithm 41

7 Cache-aware request routing algorithm for CEPH-based datalakes 66

8 Probabilistic adaptive search 86

Chapter 1

Introduction

1.1 Motivation

With the microservices architectural model, complex systems are built as a set of small loosely-coupled independent components. Using this model provides multiple advantages for developers, such as improving the maintainability by allowing to isolate modules from a complex application, therefore reducing the required effort for debugging, changing the behaviour of existing services, or adding new functionalities [55].

Deployment of microservice-based applications takes advantage of cloud computing platforms to provide higher levels of scalability, and to facilitate the deployment and operation of applications without the need to acquire and manage physical infrastructure [17].

By definition, microservices are very small modules that provides a specific service, and their minimalist design makes them suitable for lightweight virtualization (like containers) or serverless platforms (like Function-as-a-Service), which are actively promoted by public cloud providers, such as AWS Lambda[1], IBM Cloud Functions[2], Microsoft Azure[3] and Google Cloud Functions[4] [115, 150, 151].

However, the adoption of cloud platforms to operate complex applications presents particular challenges, being response times a key aspect in the decision making process [106]. Both, containers and serverless platforms, are provisioned on-demand under pay-per-use cost models, and services need to be initialized in order to be used, increasing response times, and affecting the ability to meet established service levels (SLOs) [11, 150, 24, 79, 39]. This is a limitation that prevents applications that are sensitive to response times from being migrated to the cloud [11].

A common response to performance challenges in cloud-based environments consists on resource

[1]https://aws.amazon.com/lambda/
[2]https://www.ibm.com/cloud/functions
[3]https://azure.microsoft.com/en-us/services/functions/
[4]https://cloud.google.com/functions

over-provisioning, meaning to reserve larger amounts of resources, or to keep idle services running, waiting for incoming requests. But this approach leads to resource waste and increasing operational costs [128], and represents an open research problem to be solved to increase the adoption of the aforementioned platforms.

In this thesis, we work towards improving the performance of microservices applications running on containerized and serverless cloud computing platforms through smart task scheduling decisions.

1.2 Research goals

The main goal in this research, is to identify causes of performance degradation of microservices deployed on cloud platforms, and to study how to intelligently schedule tasks, in order to improve performance metrics of microservices in FaaS and container-based architectures. The specific research goals are:

- Identify causes of performance degradation of microservices deployed on cloud platforms.
- Identify dependencies, direct or indirect, among the desired goals of a microservices scheduling algorithm.
- Propose and evaluate changes in current scheduling algorithms used by orchestration tools, to improve the performance of microservices.

1.3 Research questions

This research work seeks to answer the following research questions:

RQ1 Can we reduce launch time via intelligent scheduling decisions that seek to minimize control plane and data plane communications?

RQ2 Can we improve microservice performance by considering current node resource usage in the scheduling and placement decisions?

RQ3 How can we reconcile possibly conflicting goals of the microservices scheduling process? (e.g., balance worker load, maximize data locality, maximize code locality, meet QoS guarantees)

RQ4 Is it possible to extend the use of smart scheduling decisions to improve the performance of other components of cloud computing platforms (e.g., storage, and caching systems), to support the deployment of microservices based applications?

In addition to the study of how to improve the scheduling of microservices, the work in this thesis has led us to seek other ways in which microservices can themselves support the implementation of

self-adaptive cloud services that can lead to improved application performance. In this context, we also consider the following research question.

RQ5 How can lightweight microservices support the dynamic optimal configuration of cloud platforms components, and as a result help maintain application performance in response to changes in the workload?

1.4 Main contributions

This doctoral dissertation contributes to different areas in the understanding of the challenges that cloud platforms face in order to timely and efficiently deploy microservices-based applications, as well as a set of proposed solutions to overcome them. This section details such contributions as follow:

1.4.1 Fast deployment and execution of cloud functions in Function-as-a-Service (FaaS) platforms

Cloud function latency can be reduced via increasing code locality, in this thesis we proposed a scheduling algorithm for FaaS platforms, that seeks to improve microservices performance by maximizing cache affinity while actively avoiding worker overload. The proposed algorithm co-locates functions that share dependencies in their runtime environments on the same job node. An important feature of this algorithm is that it also keeps the load fairly balanced between all the available nodes, avoiding the presence of hot spots. This algorithm was implemented for the open source FaaS platform OpenLambda, the code is publicly available in a Github repository [5], and its design was peer-reviewed and presented in the 2nd Workshop on Hardware/Software Techniques for Minimizing Data Movement [145]. The multiple aspects of this contribution has been peer-reviewed and published in the Companion Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering [2], and in the Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) [20].

1.4.2 A closed-loop system to maintain microservices performance in containerized cloud platforms

Under a microservices architecture, applications can also deploy their components in containers managed through container orchestration systems. We found that workload spikes and poor container isolation often result in performance degradation over the lifetime of the container. Therefore in this work we proposed a design for a closed-loop system to monitor container's performance and carry

[5] `https://github.com/gtotoy/olscheduler`

out an scale-out on the right set of nodes, based on resource usage levels on worker nodes. This contribution has been peer-reviewed and published in the 5th International Workshop on Container Technologies and Container Clouds [32].

1.4.3 Smart scheduling beyond serverless-based microservices

To demonstrate that our affinity scheduling approach for serverless microservices has important applicability beyond the microservices domain, we explore the extension of the use of smart scheduling decisions to improve the performance of other components of cloud computing platforms. We selected the use case for data lakes, storage solutions that have emerged to optimize the performance of analytical tools such as Hadoop or Spark. In this thesis, we showed that the use of algorithms that prioritize maximizing the cache hit rates, without neglecting load balancing, as a feasible method to obtain better access times to the data stored in the data lake.

1.4.4 FaaS-based architecture to optimize cloud component settings

We make a case for adopting serverless-based microservices to support the dynamic implementation of self-adaptive cloud services. We designed a low cost autonomic controller to off-load the computational work required to calculate optimal configuration for cloud services, and showed the feasibility of our approach through implemeting SPREDS, a real implementation of our design using Redis and AWS Lambda. This contribution has been peer-reviewed and published in the MDPI Computers journal [35].

1.5 Regional impact and technological transfer

In order to link the development of this research, with local social needs, we have carried out some activities interacting with the local industry, seeking to evangelize the benefits of cloud platforms and to help technology transfer.

- We carried out a survey-based study to find out the reality of Ecuador regarding the use of cloud computing services [9, 34]
- We partnered with a local Software-as-a-Service (SaaS) company to develop a model and a process that can be used to properly plan the yearly cloud computing budget for a service [33].
- We studied the workload of an Online Invoicing application with clients in the Andean region in South America, in order to use it when evaluating microservices architectures [16].
- We studied a set of syllabi of distributed systems courses from top Computer Science programs around the world, to both describe current trends in teaching distributed systems and as a reference for educators that seek to improve the quality of their syllabi [6, 4].

1.6 Dissertation outline

This dissertation is structured as follows: Chapter 2 describes the related background in microservices, serverless computing, and containers. In Chapter 3, we propose an algorithm to better take advantage of cache components in FaaS platforms in order to improve cloud function latency, and we evaluate this algorithm with simulations. Chapter 4 presents the feasibility of a real world implementation of our proposed algorithm in OpenLambda. In Chapter 5, we compare the proposed algorithm with a state of the art load balancing algorithm, and measure the improvements on cloud function latency. Chapter 6 explores the use of node resource usage in the scheduling decisions for containerized platforms, and studies the benefits of performance-aware deployment of containers to improve microservices performance. Chapter 7 describes the use of smart scheduling decisions to improve the performance of a cloud based storage system, to support the deployment of microservices based applications. In Chapter 8, we study other ways in which microservices can themselves support the implementation of self-adaptive cloud services, and propose a FaaS-based architeture to support the dynamic optimal configuration of cloud platforms components. Chapter 9 summarizes the main findings in this thesis and points out directions for future research.

Chapter 2

Background

In this dissertation, our main interest is to work around the deployment of microservices in cloud computing platforms. In this chapter we provide the concepts and background required in the core chapters. In Section 2.1, we present basic concepts about cloud computing, and its main on-demand highly scalable platforms. Section 2.2, includes a definition of microservices, the preferred deployment platforms, and the challenges that are tackled in this work.

2.1 Cloud computing

Cloud computing is a term to define a current wave of computational infrastructure provisioning, in which an, usually massive, amount of computing and storage resources are shared among multiple users [130]. Public cloud computing providers, like GCP[1], AWS[2], and Azure[3], provide the users with a ever-growing amount of services, and an easy-to-use interface to encourage the adoption of these platforms.

An appealing characteristic of some tools available in cloud platforms is the *on-demand* nature. This means that resources are only provisioned, and billed, when they are effectively required for the users. The most prominent types of on-demand cloud computing platforms are serverless, and containers.

2.1.1 Serverless computing (FaaS)

Serverless computing is an emerging cloud architecture model where developers can focus on programming their applications instead of worrying about infrastructure concerns. In this model, abstraction reaches the server level, and developers can stop thinking about VM deployment, scalability,

[1]https://cloud.google.com/
[2]https://aws.amazon.com/
[3]https://azure.microsoft.com/

fault tolerance and other infrastructure issues [151].

A Function-as-a-Service (FaaS)[4] cloud platform supports the creation of distributed applications composed by a number of small, single-task, cloud functions. These functions run in lightweight sandbox environments, which run on top of virtual machines. The sandboxes, runtime environments, and virtual machines are managed by the cloud platform. Thus, a developer can create elastic cloud applications without having to worry about server provisioning and elasticity managers. Examples of FaaS platforms include OpenLambda[5], Fission[6], OpenWhisk[7], AWS Lambda[8] and Azure Functions[9].

OpenLambda

OpenLambda is serverless computing platform that supports the FaaS execution model [79]. Figure 2.1 shows the OpenLambda architecture. In OpenLambda, a developer must upload the code of her cloud functions to the *code store* or registry. When a user triggers the execution of cloud functions, a request is sent to the *load balancer*, which selects a *worker* based on the configured load balancing mechanism. When a worker receives a request, it runs the cloud function in a *sandbox*, currently implemented with Docker containers. The first time a function runs on a worker, the worker has to contact the code store to get the code of the function; the code is cached so that this step is not needed in future invocations.

Function scheduling in OpenLambda

In OpenLambda, the function scheduling (or task of assigning cloud functions to workers) is performed by nginx, with its role of software load balancer. The load balancing methods currently supported by ngninx are [110]:

- Round-robin: requests are assigned to servers in a round-robin fashion.
- Least-connected: an incoming request is assigned to the server with the least number of active connections.
- IP-hash: uses a hash-function map all requests coming from the same IP address to the same server.

These load balancing methods distribute the load between the workers, but lack functionality to make intelligent decisions that seek to, for example, minimize data transfers between workers

[4] *Function-as-a-Service (FaaS)* is the most common form of serverless; in this thesis, we use both terms interchangeably.

[5] `open-lambda.org`

[6] `fission.io`

[7] `openwhisk.apache.org`

[8] `aws.amazon.com/lambda`

[9] `azure.microsoft.com/en-us/services/functions`

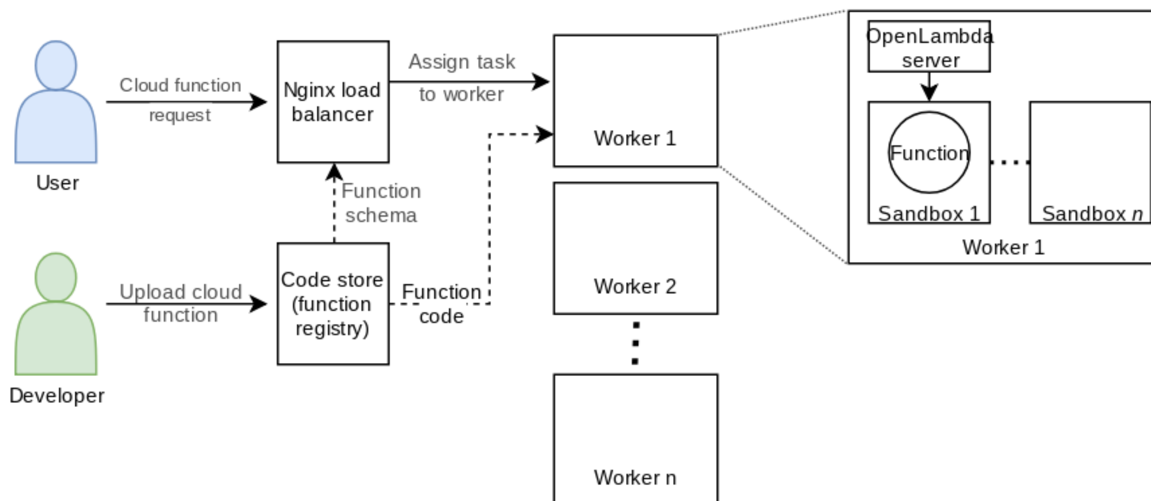

Figure 2.1: The OpenLambda architecture [79].

or between a worker and an external repository (e.g., a distributed file system or a repository of packages required by the cloud functions).

Package caching with Pipsqueak

Oakes et al. [111] proposed Pipsqueak, a shared package cache available at each OpenLambda [79] worker. Pipsqueak seeks to reduce the start-up time of cloud functions via supporting lean functions

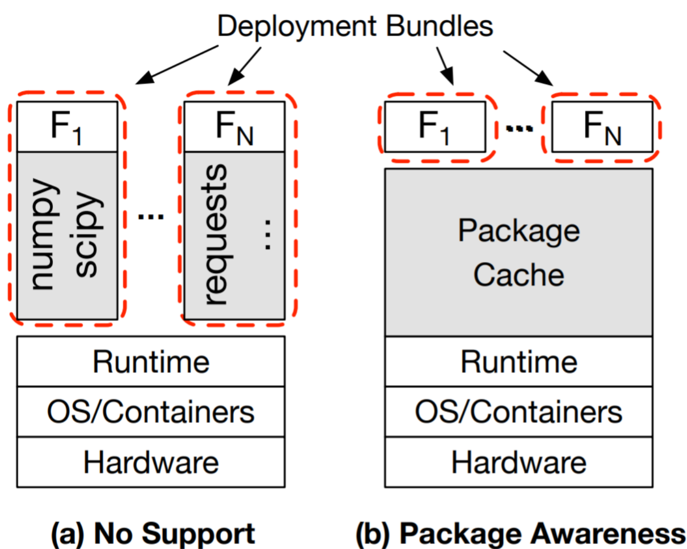

Figure 2.2: The Pipsqueak package cache in the OpenLambda stack. Figure reproduced from [111].

whose required packages are cached at the worker nodes (see Figure 2.2). The cache 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 A *and* B.

Having pre-initialized packages in sleeping containers speeds up function start-up time because this eliminates the following steps present in an unoptimized implementation: (1) downloading the package, (2) installing the package, 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 and as much as $12.8s$ for a large library like Pandas [112].

2.1.2 Containers

Containers are a lightweight virtualization technology, where the process isolation is performed at the operating system level, unlike traditional hypervisor-based solutions which virtualize at the hardware level [103]. Containers facilitate packaging, distribution, and orchestration of applications [115]. However, to guarantee short response times, containers are required to be active and waiting for connections, leading to over-provisioning of resources.

Container placement challenges

Datacenters that support cloud platforms with dynamically-requested resources need mechanisms for on-demand co-location of workloads in the same physical machines while meeting tenant service level objectives (SLOs). This is the **workload placement** problem of mapping virtual resources to physical resources and corresponding realization in the datacenter infrastructure [18]. This problem has been studied in the context of multi-tier web application placement [140], virtual machine (VM) placement [153, 18, 25], application placement [53] and placement of jobs composed of tasks [152, 81].

Each case of the workload placement problem deals with its own requirements and restrictions, given by the infrastructure and workload characteristics. For the case of services running on containers on a cluster of VMs, the amount of resources is fixed (and controlled by the VM placement layer); smart placement can lead to improved launch and service times; and, the limited isolation complicates the task of meeting tenant SLOs.

Kubernetes

Kubernetes is the most used container management tool [19], and provides cluster creation tools like Kubectl[10] and Kubespray[11]; however, their solutions may not satisfy the requirements of all users. KubeNow [41] is a tool oriented to the scientific community, which uses container clusters for data analysis, can easily run clusters on different platforms. However, it is solely focused on the ease and reduction of the cluster creation time, and does not address any functional or performance improvements in the operation stage of the cluster.

Early work on (Linux) **container scheduling**—done for the context of distributed processing platforms like Apache Spark—looked into the case of a very large system in which the scheduler must be distributed [81] and also sought to provide support for very short-lived tasks that require scheduling decisions to be extremely fast [114]. Furthermore, the main performance goal applicable to these platforms is job turnaround time, and not request service time as in the problem studied in this dissertation.

Kubernetes, supports placement restrictions based on affinity and anti-affinity, resource requirements, and user defined labels. This information is used to filter nodes during a first phase. In the second phase, the scheduling algorithm ranks nodes according to priority policies with the following goals[12]:

- Spreading service to improve reliability (SelectorSpreadPriority, ServiceSpreadingPriority).
- Preferring or avoiding nodes based on user-provided rules (InterPodAffinityPriority, NodePreferAvoidPodsPriority, NodeAffinityPriority, TaintTolerationPriority, CalculateAntiAffinityPriority).
- Seeking to balance load or CPU/memory consumption (BalancedResourceAllocation, LeastRequestedPriority).
- Bin packing (MostRequestedPriority, RequestedToCapacityRatioPriority).
- No special priority (default): Round robin (EqualPriorityMap).
- Reduce dependency load time (ImageLocalityPriority).

While these algorithms can be used to meet goals related to resource usage, load balancing and reliability, only one of them considers performance (ImageLocalityPriority), and none is suitable for applications that seek to meet tight performance SLOs.[13]

[10] https://kubernetes.io/docs/reference/kubectl/kubectl/
[11] https://github.com/kubernetes-incubator/kubespray
[12] `https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/`
[13] Parallel to this work, IBM released SSX, a scheduler expansion for k8s that avoids overloading the workers, considering actual resource usage instead of just the amount of allocated resources. We discuss SSX in section 6.5.

2.2 Microservices

2.2.1 Definition of microservices

Microservices are an architectural model for software development, derived from service-oriented architecture (SOA), which allows building an application as a set of independent processes, collaborating with each other through messages. This architecture of independent but interconnected processes allows the isolation of modules from a complex application, making easier the debugging of errors, and the addition of functionalities [64, 55].

Adoption of microservices architecture has increased driven by (1) the boost it gives to maintainability, which support modern agile development methodologies, and (2) the success of their implementation by industry giants such as Netflix [143], Amazon [72], Soundcloud [40], e-Bay and Google [133]. Their application scope is highly fragmented [54], with a broad range of implementations, such as Internet of Things (IoT) [93, 38, 90], scientific research [59, 49, 160], business process modeling [12], and chatbots [166].

2.2.2 Platforms for the deployment of microservices

Being a set of independent, coordinated processes, microservices benefit from elasticity and automated deployment features available in cloud computing services [54, 148]. Containers, a lightweight virtualization technology available in the cloud, and serverless computing platforms, where a small stateless portion of code is executed in response to an event, are the most popular platforms for deploying microservices because they ease the packaging, distribution, and orchestration of microservices [115, 23, 150, 151].

2.2.3 Challenges for the deployment of microservices in the cloud

As mentioned before, the application scope for microservices is extensive, leading to increasing levels of adoption. However, some applications have higher requirements in terms of response times, low latency, and responsive scalability [52], which can be negatively affected when concatenated microservices amplify any bottleneck effect along the end-to-end user experience [66]. These kind of applications presents several challenges for the cloud platforms selected to deploy a cloud microservices architecture [24, 78, 13, 97, 149, 61]. *Systems-related* factors influence the application performance, across two dimensions: (i) *run-time performance*, and (ii) *initialization times* of microservices.

Performance isolation is a main challenge facing the deployment of microservices on cloud platforms, especially in those public clouds where multi-tenancy policies share the physical server resources between different customers. A "*noisy neighbor*" can affect both the initialization time and the run-time performance of other clients hosted on the same physical server. Even though the

control of the resources assigned to each container is quite accurate, there are still problems to solve, such as the unaccounted CPU usage resulting from network transmission level operations [89] or the interference caused by high loads at the access level to storage media [161]. The community is mostly addressing this challenge via ensuring isolation at the operating system level [89, 161, 96, 73], through the work in this thesis, we've shown that this problem can also also be tackled through proper monitoring and scheduling of containers [32].

Scheduling and placement of microservices, within the platform on which they are deployed, present some of the greatest research challenges. The placement of the microservices in container-based and serverless frameworks is generally based on simple concepts like bin packing (e.g., in Kubernetes) and load balancing (e.g., in OpenLambda and OpenWhisk, and the spread policy in Kubernetes). In some cases, migrating microservices between nodes is included to ensure balance [87]. Other planners use some kind of prioritization, such as labels and *constraints* [152], the size of jobs to be executed [69], or a combination of load balancing, applications performance, and network overhead [74]. In either case, initialization latency is not considered a priority, although it is mentioned as a desirable feature [69].

Cold start is another important challenge for cloud based microservices. This problem is present in both containers and serverless platforms, and affects directly on the time it takes for a microservice to run and start serving orders (e.g., when a microservice is under a heavy workload and requires additional instances to meet the expected service level). Operators of container-based plaforms mitigate this issue by keeping virtual machines up and available for hosting containers, but this is a cost burning aspect to be kept under control [33]; we show that using performance-aware placement can minimize the initialization time for container-based microservices [32].

Due to their event-based execution nature, serverless platforms may require to initialize a runtime environment where the cloud functions will be executed. This initialization phase induce possible increases in latency or response times for the microservices in serverless platforms [11]. Previous attempts to address this problem include using a cache system to reduce cold boot times by storing recently used runtime environments [111], or by loading pre-packaged libraries [31].

Chapter 3

Package-aware scheduling of FaaS functions

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.

The work presented in this chapter was first published at the HotCloudPerf Workshop co-located with the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE'18) [3].

3.1 Introduction

The *serverless computing* paradigm [79, 150] 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 [150]. 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 [149]. 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 [79, 111, 149]: (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 [111][1]. Moreover, long function launch times have a direct negative impact on the performance of serverless applications using the FaaS model [149]. 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) [112]. 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 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 3.2. The proposed algorithm aims to maintain a good balance between maximizing cache affinity and minimizing the node imbalance.

[1] For simplicity, in this work 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 [111, 112].

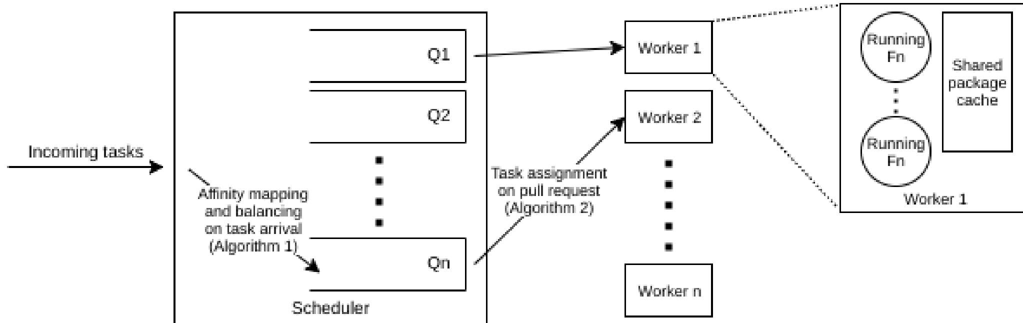

Figure 3.1: Model assumed in the algorithm proposed in section 3.2.3. The scheduler assigns incoming tasks to queues, based on package affinity while avoiding node imbalance (design goals). The worker nodes have a shared package cache that can be leveraged by the cloud functions to speed up the startup times. Variations of the algorithm suitable for a push-based scheduling, distributed scheduling and multi-package affinity are discussed in sections 3.2.4, 3.2.5, and 3.2.6, respectively.

2. Besides the proposed algorithm for pull-based scheduling, we identify potential extensions for alternative forms of scheduling: push-based scheduling in section 3.2.4, distributed scheduling in section 3.2.5, and how to extend the algorithm to deal with multiple packages in section 3.2.6.

3. In section 3.3 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.

3.2 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.

3.2.1 System model and assumptions

In section 3.2.2 we outline the goals for a package-aware scheduler for FaaS functions; our proposed scheduling algorithm is described in section 3.2.3. 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 3.1, 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 3.2.4 and for distributed scheduling in section 3.2.5.

Package caching We assume that there is a package or library caching mechanism implemented at the worker level; as described in section 2.1.1. 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 2.1.1.

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 3.2.6 we discuss how to extend the algorithm to consider multiple package requirements.

3.2.2 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 [98]. Better alternatives are Join-the-Shortest-Queue (JSQ) [75] and Join-Idle-Queue (JIQ) [98].

To **maximize cache affinity** (of the package cache), we can use consistent hashing [88] 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 chapter, *our goal is to maintain a good balance between maximizing cache affinity and minimizing the node imbalance.*

3.2.3 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 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 [105],

[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.

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
```

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.

3.2.4 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 3.2.3 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 [75].

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.

3.2.5 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 [27]. 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 [98]. 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 [98], which decouples discovery of lightly loaded servers from job assignment, thus leading to very fast task assignments.

3.2.6 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.

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

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 2.1.1 and 3.2.1); we leave relaxing this assumption for future work.

3.2.7 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 3.3, 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.

3.3 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 $0.1ms$.

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

- 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 [79, 111, 97].

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 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 3.2.6), and a per-worker package cache.

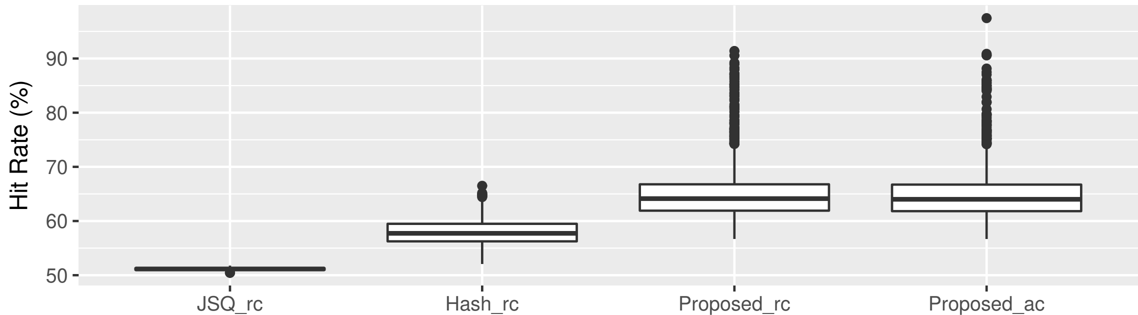

Figure 3.2: Box plots of the hit rates of the package caches at the worker nodes. Our algorithm improves the average hit rate by actively seeking to improve package-affinity.

Table 3.1: Task latency percentiles (in seconds). Our algorithm improves latency due to improved cache hit rate.

Algorithm	$50th$	$90th$	$95th$	$99th$
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

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 3.2.7).

Our preliminary **results** show we can improve the median **hit rate** from 51.2% (JSQ_{rc}) to 64.1% ($Proposed_{rc}$), as shown in Figure 3.2. The proposal to cache only affinity packages ($Proposed_{ac}$) produced results very similar to those of $Proposed_{rc}$.

The improved hit rate has a positive effect on the **latency** of the tasks, as shown in Table 3.1. Median latency improves by 65.6% ($Proposed_{rc}$ vs. JSQ_{rc}), and tail latency improves by 40.5% ($90th$ percentile). Note that we avoid the straw man fallacy of comparing our algorithm against JSQ with no caching, as this is an unfair comparison. Both JSQ_{rc} and $Proposed_{rc}$ are much better than 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 3.2[4]. We can observe that our algorithm sacrifices some unbalance, to seek a higher hit rate (and smaller latency). JSQ achieves near perfect balancing, while the hash-based affinity algorithm produces the most unbalanced task assignments.

[4]This simple metric quantifies the dispersion in the total work assigned to each worker.

Table 3.2: Node unbalance, measured using the coefficient of variation of the number of functions assigned to each worker (smaller is better).

Scheduling Algorithm	c_v
JSQ_{rc}	1.02
$Hash_{rc}$	357.65
$Proposed_{rc}$	65.33
$Proposed_{ac}$	66.06

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.

3.4 Related work

We build upon a large body of work in task scheduling. Early work in affinity scheduling sought to improve performance in multi-processor systems by reducing cache misses via preferential scheduling of a process on a CPU where it has recently run [144, 62]. 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 chapter, is the case of locality- or content-aware request distribution in Web-server farms [42]. 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 [116, 42]. 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 [43]), 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 [75] and distributed load balancing [105]. For example, the Join-Idle-Queue (JIQ) algorithm incurs no communication overhead between the dispatchers and processors at job arrivals [98]; 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 3.2.5).

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 [162]; 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 [168, 158, 163]. However, these are not directly applicable to the problem studied in this chapter, 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 [111, 63, 131]. In particular, we were inspired by the recent work in caching packages in OpenLambda by Oakes et al. [111], though they left the global scheduling work (to improve cache hit rates) for future work.

3.5 Conclusions

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.

Chapter 4

Package-aware task scheduling in real serverless platforms

In a distributed computing platform, co-locating tasks at worker nodes that store or cache any required files is a time-proven mechanism to reduce task latency. While this problem has been extensively studied in the Web and Big Data processing domains, it is only recently gaining attention in the serverless computing domain. One proposed optimization for Function-as-a-Service (FaaS) clouds is to cache required packages at the worker nodes instead of bundling them with the cloud functions, thus significantly reducing the function launch time. However, existing FaaS schedulers are vanilla load balancers that do not attempt to minimize the movement of packages or files across the network. As researchers start tackling the problem of *package-aware scheduling* and other near-data scheduling optimizations for FaaS platforms, having a common framework on top of which to implement and evaluate their ideas would be beneficial, as this would encourage fair comparisons between different solutions and facilitate experiment reproducibility. To address this problem, we present a simple and extensible function scheduler for the OpenLambda FaaS platform. Our scheduler is implemented in Go, and is simpler to modify and extend than the ngninx load balancer used by the original OpenLambda. To illustrate the usefulness of our scheduler, we added a package-aware scheduling algorithm to it. We have released our code so that others can easily implement new scheduling algorithms for OpenLambda.

The work presented in this chapter was first published at the 2nd Workshop on Hardware and Software Techniques for Minimizing Data Movement (Min-Move) co-located with the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018) [145].

4.1 Introduction

Functions-as-a-Service (FaaS) cloud platforms enable tenants to deploy and execute functions on the cloud, without having to worry about server provisioning. In a FaaS platform, *cloud functions* are (typically) small, stateless tasks, with a single functional responsibility, and are triggered by events. The FaaS cloud provider manages the infrastructure and other operational concerns, enabling developers to easily deploy, monitor, and invoke cloud functions [150]. These functions can be executed on any of a pool of servers managed by the provider and potentially shared between the tenants.

One proposed optimization for FaaS platforms is to cache required packages or libraries at the worker nodes instead of bundling them with the functions, thus making the functions more lean and as a result, significantly reducing their launch time [111, 3]. However, existing FaaS schedulers are vanilla load balancers that do not attempt to maximize package-locality when assigning functions to workers. For this reason, the development of new scheduling algorithms for FaaS platforms is considered an important challenge in the serverless computing domain [79, 150, 149, 3].

As researchers start tackling the problem of *package-aware scheduling* and other near-data scheduling optimizations for FaaS platforms, having a common platform on top of which to implement and evaluate their ideas would be beneficial, as this would encourage fair comparisons between different solutions, facilitate experiment reproducibility and reduce development time.

Towards this goal, we have implemented `olscheduler`, a simple and extensible function scheduler for the OpenLambda FaaS platform. Our scheduler has 331 lines of Go code, and is simpler to modify and extend than the ngninx load balancer used by the original OpenLambda. `olscheduler` comes with three scheduling policies (random, round-robin and least-loaded), and exposes useful information about the platform to the system developer, so that additional scheduling policies can be easily implemented.

To illustrate the usefulness of `olscheduler`, we added a simple package-aware scheduling algorithm that seeks to improve package-affinity while avoiding unmanageable worker overload. We were able to implement this algorithm with only 46 additional lines of code (LOCs). Preliminary simulation results show that this simple approach can cut the function latency by more than 65%. In the future, we will validate our results in real cloud experiments.

We have released our code so that others can easily implement new scheduling algorithms for OpenLambda[1].

4.2 Design and implementation

We decided to implement `olscheduler` using Go, as its primitives lets the developer easily build high-performant distributed systems.

[1] `https://github.com/gtotoy/olscheduler`

To facilitate the implementation of future schedulers, `olscheduler` exposes functions and data structures that can be used to get the following information:

1. Workers: Number and references to the workers.
2. Per-worker load: Measured as the number of active requests that each worker is currently handling.
3. Required packages: The list of required packages, sorted by size, is exposed for each function call.
4. Function schemas: Obtained by querying the code repository and cached in memory.

To get the number of required packages (item 3 above), we extended the HTTP Post request (cloud function request in Figure 2.1) so that it supports receiving the list as an annotation on the function call. An alternative design would have been to query the cloud store to get this information; we rejected this idea to avoid an extra step on the critical path of the function requests.

`olscheduler` currently supports the following scheduling algorithms:

- Round-robin: Distributes the requests uniformly between the workers, in round robin fashion.
- Least-loaded: An incoming request is assigned to the worker that currently has the least number of active requests.
- Random: Distributes requests randomly between the workers, according to user defined worker weights.

The functionality described above was implemented in 331 lines of code (LOCs). In addition, we have implemented an additional, package-aware scheduling policy, as described next.

4.2.1 Package-aware scheduling in olscheduler

To illustrate that `olscheduler` can be easily extended to implement a package-aware algorithm, we implemented a variant of an algorithm we proposed in prior work [3], adapting it so that it is suitable for the OpenLambda scheduler model: a push-based, centralized, scheduler with a processor-sharing service discipline (workers).

The algorithm we implemented seeks to maximize cache affinity (with respect to the packages in the package cache), while avoiding overloading workers beyond a configurable threshold. The scheduler 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 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 number of active requests has

Algorithm 4: Package-aware scheduler algorithm for OpenLambda

```
   Global data: List of workers, W = w_1, ..., w_n, Hash functions H_1 and H_2, maximum load
                threshold, t
   Input: Function, f, list of required packages sorted by descending package size,
          P = p_1, ...p_n
1  if (P is not empty) then
       /* Greedily seek affinity w/ large package                          */
2      for (l = 1, ..., |P|) do
           /* Calculate two possible worker targets                        */
3          t1 = H_1(p_l)%|W| + 1
4          t2 = H_2(p_l)%|W| + 1
           /* Select target with least load                                */
5          if (load(w_t1) < load(w_t2)) then
6              A := t1
7          else
8              A := t2
           /* If target is not overloaded, we are done                     */
9          if (load(w_A) < t) then
10             Assign f to w_A
11             return
   /* Balance load                                                         */
12 Assign f to least loaded worker, w_i
```

exceeded a threshold, t), then the scheduler chooses the worker with the least load. To improve cache affinity while improving load balance, we apply the power-of-2 choices technique [105], by using two hashing functions to map a task to a worker; each hash function maps the task to a different worker, and the task is assigned to the least loaded one. Algorithm 4 shows our package-aware scheduling algorithm for OpenLambda.

With the information exposed by `olscheduler`, implementing this policy was straightforward, and took only 46 LOCs.

4.2.2 Validation results

We have performed validation tests to make sure our scheduler works according to our design. We implemented both the least-loaded and the proposed package-aware algorithm in a simulator, with the following configuration parameters:

- Mean inter-arrival time = $0.1ms$ (exponentially distributed).
- 1 000 worker nodes; each can run up to 100 functions simultaneously ($st = 100$).
- Distribution of packages popularity: Zipfian ($s = 1.1$).

- Time to start the packages is exponentially distributed (average time to start = $4.007s$).
- Number of packages required by a function is exponentially distributed (mean required packages = 3).
- 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: $t = 200$.

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 [79, 111, 97].

Our preliminary **results** show we can improve the median **hit rate** from 51.15% (least-loaded) to 63.52% (Proposed). This has a direct impact on latency, as shown in Table 4.1: median latency improves by 65.8% (Proposed vs. least-loaded), and tail latency improves by 41.9% ($90th$ percentile). If we compare against a least-loaded load balancer in an unoptimized platform that does not cache function packages, our algorithm improves median latency by 189.9 times.

Table 4.1: Task latency percentiles (in seconds). Our algorithm improves latency due to improved cache hit rate.

Algorithm	$50th$	$90th$	$95th$	$99th$
Least-loaded, no pkg cache	256.42	455.76	480.71	503.72
Least-loaded	3.95	18.53	25.29	40.86
Proposed	1.35	10.76	15.90	28.98

4.3 Related work

We build upon prior work in load balancing for server clusters [105, 75, 98]. Most of this work is specific to the Web server farms, though these balancing algorithms are easy to extend and apply to FaaS architectures. In the FaaS domain, industry schedulers—like the generic nginx and the custom function schedulers for OpenWhish, Fission–are vanilla load balancers that do not seek to minimize data movement in the system.

Another line of research is work in task scheduling while trying to improve data-locality, for example, for the Web [116, 43, 42] and Big Data processing [168, 162, 158, 163] domains.

The near-data scheduling problem arises in frameworks like Hadoop, where each type of task has different processing rates on different subsets of servers [162]; 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 [168, 158, 163]. One thing that has helped the community propose new schedulers for Hadoop—some of which have later made it to the Hadoop codebase, like [168]—is the fact that Hadoop's design makes it easy to replace the scheduler with a new algorithm.

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 [111, 131, 149].

4.4 Conclusions

Function-as-a-Service platforms—like OpenWhisk, OpenLambda and Fission—could benefit significantly from intelligent schedulers that seek to minimize data transfers, as this can be an expensive step (e.g., reading input data from a distributed file system, or downloading and installing packages from a package repository). In this work, we presented a simple yet extensible function scheduler for OpenLambda, which can be used as a basis for future research in smart scheduling for FaaS platforms. In the next chapter we use our scheduler to evaluate different package-aware scheduling algorithms.

Chapter 5

Package-aware task scheduling versus traditional load balancing algorithms

Fast deployment and execution of cloud functions in Function-as-a-Service (FaaS) platforms is critical, for example, for microservices architectures. However, functions that require large packages or libraries are bloated and start slowly. An optimization 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 packages cached in response to prior function executions, and cannot properly reap the benefits of package caching. We study the case of *package-aware scheduling* and propose PASch, a novel scheduling algorithm that seeks package affinity during scheduling so that worker nodes can re-use execution environments with preloaded packages. PASch leverages consistent hashing and the power of 2 choices, while actively avoiding worker overload. We implement PASch in a new scheduler for the OpenLambda framework and evaluate it using simulations and real experiments. When using PASch instead of a least loaded balancer, tasks perceive an average speedup of 1.29x, and 80th percentile latency that is 23x faster. Furthermore, for the workload studied in this chapter, PASch outperforms consistent hashing with bounded loads—a state-of-the-art load balancing algorithm—yielding a 1.3x average speedup, and a speedup of 1.5x at the 80th percentile.

The work presented in this chapter was first published at the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2019) [20].

5.1 Introduction

Function-as-a-Service (FaaS) cloud platforms let tenants deploy and execute functions on the cloud, without having to worry about server provisioning. In a FaaS platform, cloud functions are typically small, stateless tasks, with a single functional responsibility, and are triggered by events. The FaaS provider manages the infrastructure and other operational concerns, enabling developers to easily deploy, monitor, and invoke the functions [150]. These functions or *tasks* run on any of a pool of servers or *workers* managed by the provider and potentially shared between the tenants.

The FaaS model holds good promise for future cloud applications, but raises new performance challenges that hinder its adoption [149]. One of these challenges is reducing the FaaS overhead. In particular, the provisioning overhead that results from deploying the cloud function on demand as part of unpredictable workloads, can make launching functions slow, and as a result, out-of-the-box exceed service level objectives (SLOs) of applications from performance-critical domains like interactive web applications and IoT environments.

Cloud functions can launch rapidly, as they run in preallocated virtual machines (VMs) and containers. However, when these functions depend on large packages, they start slowly; this affects the elasticity of the application, as it reduces its ability to rapidly respond to sharp load bursts [111][1]. Moreover, long function launch times have a direct negative impact on the performance of serverless applications using the FaaS model [149]. A solution is to cache pre-imported 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 [112].

However, existing FaaS schedulers are agnostic of any package caching implemented at the worker nodes and—when assigning functions to workers—fail to properly leverage the cached packages, as illustrated in Figure 5.1.

In this chapter, we study the case of *package-aware scheduling*, which we define as a special case of near-data scheduling optimizations for FaaS platforms.

Existing FaaS schedulers, like those in OpenWhisk, Kubeless, Fission, OpenFaaS, and OpenLambda, are simple load balancers that make no attempt to target any code or artifacts cached at the worker nodes. With a regular load balancer, a function can be mapped to a worker node that does not contain a required package, even though one or more worker nodes that do have the package may be available (see Figure 5.1). A solution could be to use available techniques like consistent hashing [88, 138] to route all function requests that require a specific package p to the same worker node, thus maximizing the cache hit rate. However, this approach suffers from a problem of load imbalance, particularly under skewed workloads [109]—as is the case of the distribution of packages required by functions in FaaS platforms (see Table 5.1 and [111]).

Within the domain of stream processing engines and the general *balls and bins* model, recent work

[1] For simplicity, we talk about *large* packages, but the start-up time includes the time to download and install the package, and the run-time import processes; on average, all this can take more than four seconds [111, 112].

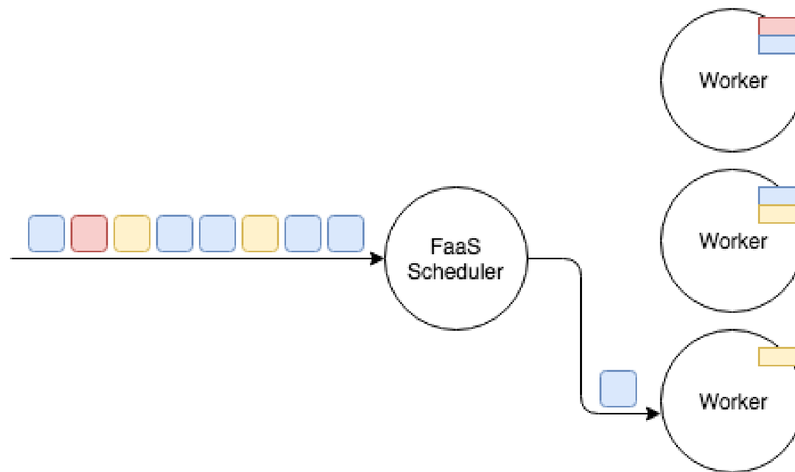

Figure 5.1: Example of function execution requests arriving at a FaaS scheduler. The color of each request represents the largest package that the function requires. The stack on the upper right corner of the worker nodes represents the local import (package) cache, available to all functions that run on the worker. Current schedulers have a load balancing goal and make no attempt to try to schedule a function where it could run faster; in this case, that would be at one of the nodes that has already cached and pre-imported the package.

has tried to map requests to specific servers while keeping a balanced load [109, 108, 104], however these solutions avoid imbalance between the worker nodes at all costs, regardless of whether a worker could tolerate more work without sacrificing performance. We show that this is not necessary, and relax the balancing goal so that the ultimate goal is not keeping the workers balanced, but rather avoiding exceeding worker capacity.

To solve this problem, we propose a fast scheduling algorithm that seeks package affinity during scheduling, while actively avoiding worker overload. Our algorithm leverages the *power of 2 choices* technique [105] to map a task to the least loaded of two affinity nodes, based on the largest package required by the function. If, however, both nodes exceed a configurable overload threshold, then the scheduler reverts to simple load balancing, and maps the task to the least loaded worker node. To seek package affinity during task assignment, we use *consistent hashing* [88, 138] so that we minimize the expected number of movements in case worker nodes are added or removed by an auto-scaler or elasticity manager. As a result, worker nodes can re-use execution environments with preloaded packages, thus resulting in faster task launch time, and consequently, a faster task turnaround time.

Our contributions consist of the following:

1. We carefully describe the problem and related work, including state-of-the-art algorithms (section 5.2).

2. We formalize the problem of scheduling functions in FaaS platforms with the goal of maximizing code locality while actively avoiding worker overload (section 5.3).

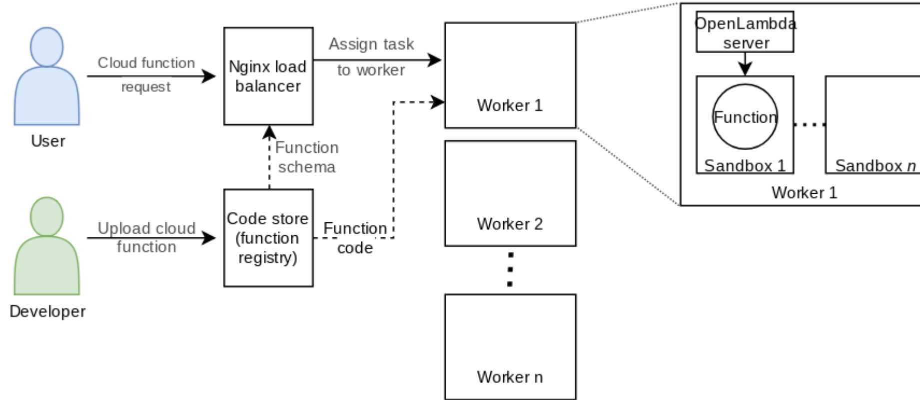

Figure 5.2: The OpenLambda architecture. The function scheduling (or task of assigning functions to workers) is performed by the NGINX software load balancer.

3. We propose PASch, a novel scheduling package-aware algorithm for FaaS platforms (section 5.4).

4. We provide a working implementation of PASch on `olscheduler`, a lightweight scheduler for the OpenLambda framework that we have released (section 5.4.2).

5. We evaluate PASch under realistic workloads (section 5.5) and find that it considerably outperforms the least loaded scheduler: 1.29x average speedup and 80th percentile in latency is 23x faster. In our experiments, PASch also outperforms a state-of-the-art algorithm [104] yielding a 1.3x faster average latency and a 80th percentile in latency that is 1.5x faster.

5.2 Background and related work

5.2.1 FaaS and OpenLambda

A Function-as-a-Service (FaaS) platform supports the creation of distributed applications composed by small, single-task, cloud functions. These functions run in lightweight sandbox environments, which run on top of virtual machines. The sandboxes, runtime environments, and virtual machines are managed by the cloud provider. Thus, a developer can create elastic cloud applications without having to worry about server provisioning and elasticity managers. Examples of FaaS platforms include OpenLambda,[2] Fission,[3] OpenWhisk,[4] AWS Lambda,[5] Google Cloud Functions,[6] and Azure

[2] `open-lambda.org`
[3] `fission.io`
[4] `openwhisk.apache.org`
[5] `aws.amazon.com/lambda`
[6] `cloud.google.com/functions`

Functions[7].

OpenLambda OpenLambda is a serverless computing platform that supports the FaaS execution model [79]. Figure 5.2 shows the OpenLambda architecture. In OpenLambda, a developer must upload the code of their cloud functions to the *code store* or registry. When a cloud function is triggered, a request is sent to the *load balancer* (scheduler), which selects a *worker* based on the configured algorithm. When a worker receives a request, it runs the cloud function in a *sandbox*. OpenLambda currently supports Docker containers and lightweight SOCK containers [113]. The first time a function runs on a worker, the worker has to contact the code store to get the code of the function; the code is cached so that this step is not needed in future invocations.

Function scheduling in OpenLambda The function scheduling, or mapping of functions to workers, is performed by the NGINX software load balancer. The request routing methods currently supported by NGINX are [110]:

- Round-robin: maps requests to servers in round robin fashion.
- Least-connected: assigns a request to the server with the least number of active connections.
- IP-hash: uses a hash-function to map all requests coming from the same IP address to the same server.

These methods distribute the load between the workers, but lack functionality to make intelligent decisions that seek to, for example, minimize data transfers between workers or with an external repository (e.g., a distributed file system or a repository of packages required by the cloud functions).

Caching to improve task launch times Oakes et al. [111] proposed Pipsqueak, a shared import (package) cache available at each OpenLambda worker. Pipsqueak seeks to reduce the start-up time of cloud functions via supporting lean functions whose required packages are cached at the worker nodes. The cache 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 multiple package dependencies, Pipsqueak uses 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 A *and* B.

[7] `azure.microsoft.com/en-us/services/functions`

Having pre-initialized packages in sleeping containers speeds up function start-up time because this eliminates the following steps present in an unoptimized implementation: (1) downloading the package, (2) installing the package, 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 time consuming, as it can take $4.007s$ on average and as much as $12.8s$ for a large library like Pandas [112].

In addition to the Pipsqueak import (package) cache, OpenLambda workers have two other caches: an on-disk (package) install cache and a handler (function) cache.

5.2.2 Load balancing and scheduling

We build upon a large body of work in task scheduling and load balancing for server clusters. In this section, we discuss the most relevant prior work.

Affinity 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 [144, 62]. 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 avoiding unfairness and respecting thread priority.

Near-data scheduling The near-data scheduling problem is a special case of affinity scheduling applicable to data-intensive computing frameworks like Hadoop[8], where each type of task has different processing rates on different subsets of servers [162]. Tasks that process local data execute the fastest, followed by tasks that require data in the same rack, followed by tasks that read remote data. Several near-data schedulers have been proposed for Hadoop [168, 158, 163]. However, these are not directly applicable to the problem studied in this chapter, 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 requires no centralized directory and has minimal overhead. Implementing a directory of cached packages in a serverless computing platform would impose significant overhead on the infrastructure due to extra communication and storage requirements. Moreover, unlike data block storage in Hadoop, the contents of an import 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.

Task Scheduling in FaaS Platforms In the FaaS domain, industry schedulers—like the generic NGINX and the custom function schedulers for OpenWhisk and Fission—are vanilla load balancers that do not seek to minimize data movement in the system. Other than earlier work from our own

[8] `hadoop.apache.org`

group [3], we are not aware of any publications in this emerging area. However, FaaS scheduling is expected to attract work in the near future, as it is an important serverless challenge [149].

Content-aware Request Routing for Web Servers Better related to our problem is the case of locality or content-aware request distribution in Web-server farms [42]. In this context, the simplest solution is static partitioning of server files using URL hashing, to improve cache hit rate; though this can lead to extreme server overload for skewed workloads [42]. Others have proposed algorithms that partition Web content to improve cache hit rate using static tables or variants of hash-based affinity [116, 42] that are inadequate for the skewed and highly dynamic workloads of modern cloud systems. Moreover, some of these solutions assumed that the workloads are relatively stable. However, modern cloud workloads are more dynamic, making solutions that require offline workload analysis (e.g., see [43]), inadequate for this domain.

Nevertheless, some early ideas in Web request routing and traditional load balancing can be partially applied to the problem studied in this chapter. Specifically, we leverage two classic techniques in our work: consistent hashing [88] and power of two choices load balancing [105]. We provide a brief description of these techniques next.

Consistent Hashing Consistent hashing [88, 138] is an alternative to hash-based affinity that solves the problem of having a dynamic set of servers. This technique assigns a set of items (keys, client IPs, etc.) to servers so that each one receives roughly the same number of items. However, unlike standard hashing schemes, a small change in the server set does not induce a total remapping of items to servers and instead yields only a slightly different assignment. However, this approach suffers from a problem of load imbalance, particularly under skewed workloads [105, 109, 104].

Power of two choices The power of two choices [105] is a technique that achieves near perfect load balance as follows: When a request arrives to the system, the balancer selects two random servers and assigns the request to the least loaded of those two servers. Our approach combines hash-based routing with power of two choices, so that the two servers are not chosen at random, but are rather based on the *affinity* that a server has to a specific item (package). As this can lead to an unbalanced load, we add the concept of a maximum tolerable threshold, and thus avoid saturating the workers.

Recent related work Others have recently looked into how to map items to servers while simultaneously balancing the load of the servers, even in the presence of skewed workloads and a dynamic server set [109, 108, 104].

Partial key grouping [109] and consistent grouping [108] are specific to streaming frameworks like Apache Storm. For this reason, they make the strong assumption that it is detrimental to the performance of the system to split keys between workers (strict affinity). Partial key grouping

supports splitting keys only between two workers, while consistent grouping does not support key splitting at all. In contrast, as we deal with caching we can implement soft affinity instead: if both of the affinity workers for a key (package) are overloaded, then the request can be sent to any other worker.

Consistent hashing with bounded loads [104] is framed as a generic hashing solution. Though similar to ours, the problem solved by this algorithm has stronger requirements: a deterministic algorithm for mapping keys to servers that achieves load balancing that is bounded within some ϵ. We relax both requirements as described in section 5.3.

While these recent prior work could be used to solve the problem at hand, the slight differences in the problems that they try to solve (discussed in section 5.3) yield solutions that are suboptimal (as shown in section 5.5).

In a prior, work-in-progress publication [3], we proposed a set of package-aware scheduling algorithms for FaaS platforms, spanning different scheduler models: pull-based or push-based, and centralized or distributed scheduler. Our simulation-based evaluation suggested that it is a promising area of research, as our package-aware scheduling algorithm yielded improvements in latency of 66% (versus a least-loaded scheduler). In this chapter, we propose PASch, a variant of this family of algorithms that it is suitable for the OpenLambda scheduler model: a push-based, centralized, scheduler with a processor-sharing service discipline (workers). We evaluate PASch with simulations and real experiments in AWS.

5.3 Problem definition

We consider a Function-as-a-Service platform in which *tasks* are cloud functions that are executed on *worker* nodes, as a response to user-defined events.

5.3.1 Assumptions and limitations

A worker node is capable of running multiple tasks simultaneously and can be, for example, a virtual machine managed by a container orchestration engine such as Kubernetes. In addition to worker nodes, there is a *scheduler* which assigns or maps tasks to workers. The workers use a processor-sharing service discipline in which tasks share the worker's local resources. The number of workers is dynamic; this is common in cloud platforms with auto-scalers or elasticity managers that add or remove nodes to the cluster, as a response to changes in demand.

We assume there is a package or library caching mechanism implemented at the worker level; as described in section 2.1.1. When a function depends on more than one package, we seek to achieve scheduling (soft) affinity for the largest one, as this is the package that is most useful to accelerate its loading time.

Table 5.1: Top ten required packages by Python and Java files in GitHub. We analyzed only those files that reference amazonaws (Python) or com.amazonaws.services.lambda (Java).

Python Package	%	Java Package	%
boto	27.0%	com.amazonaws	42.0%
ansible	6.3%	java.util	11.1%
os	3.7%	java.io	9.3%
tests	3.3%	com.fasterxml	4.4%
time	2.4%	org.apache	3.3%
json	2.4%	org.junit	3.2%
xmodule	2.0%	javax.annotation	2.9%
django	1.9%	org.mockito	2.2%
sys	1.9%	com.visionarts	1.5%
datetime	1.8%	com.google	0.1%
All others	47.3%	All others	20.0%

We assume package popularity in cloud functions is skewed, with a long tail of unpopular packages. To confirm our intuition, we performed an analysis of the GitHub repositories of projects that could potentially contain AWS Lambda functions. Specifically, we found Python files that contain the substring *amazonaws* and Java files that contain *com.amazonaws.services.lambda*. We analyzed the import statements of those files and found that the popularity distribution of the required packages is indeed skewed (see Table 5.1).

Finally, the scheduler is agnostic of the contents of the worker caches. We believe that having the scheduler keep track of the contents of the import caches is not adequate in the FaaS setting, as this would impose additional overhead on the system (network communications to keep information updated, and resources to store and manage the caching directory).

5.3.2 System model

Given an instance of a FaaS platform, let W be the set of workers in which tasks can run, where $|W| = n$. Each worker $w \in W$ runs on a machine with limited capacity and known execution threshold t_w. For simplicity, we assume that there is a single important resource on which machines are constrained such as memory or processing. Each worker node $w \in W$ can execute an unbounded number of concurrent tasks; however, the t_w threshold (expressed in normalized units of the constrained resource) is the maximum number of resource units that can be in use at w without reducing the performance of the tasks at w.

The input to the scheduler is a sequence of task execution requests $r = \langle i, f, p, t_i \rangle$, where i identifies the request, f is the function to execute, $p \in P$ is the largest package required by the task, and t_i is the timestamp at which the request arrives to the scheduler. The requests arrive in ascending order by t_i. Upon receiving a request r, the scheduler makes a placement decision and chooses one of the workers in W to execute r.

As tasks arrive to the system, the scheduler maps and forwards the request to one of the workers. After the task is processed, it leaves the system. We define the *task turnaround time* or latency to be the difference between the time at which a task arrives to the system (t_i) and when it leaves the system.

Given the system model and assumptions described above, we define the problem we are trying to solve as follows.

Problem Given a sequence of task execution requests, each of which requires a package drawn from a skewed distribution P, and a set of workers $w \in W$, each with limited capacity and known execution threshold t_w, find a scheduling function $S : P \to W$ that maximizes affinity to the import cache at each worker while avoiding exceeding the workers' execution thresholds.

Additional goal, stability We also consider a stability goal that seeks to contain or minimize any changes in package-to-worker affinity, when the set of workers is dynamic.

5.3.3 Naive solutions

A naive solution would be to try to either balance the load or to make placement decisions seeking strict affinity from packages to workers. However, these solutions do not consider both goals together and make suboptimal placement decisions, as briefly explained next. For more detail on these and other related work, see section 5.2.

To **balance the load**, round robin or random assignment is not optimal, as the resource consumption of tasks may vary [98]. Better alternatives are Join-the-Shortest-Queue (JSQ) [75] and Join-Idle-Queue (JIQ) [98]; however, these are not applicable when the workers use a *processor sharing* approach, as is the case in OpenLambda and Fission. When the workers use a processor sharing, the least loaded policy balances the load, for example, as implemented by NGINX (least-connected policy). However, this policy does not do anything to maximize cache affinity.

To **maximize cache affinity** with an added goal of **stability** in the presence of a dynamic set of workers, we can use consistent hashing [88, 138] to assign all tasks that require a particular package to the same worker. However, as package popularity is not uniformly distributed, this approach would create hot spots, overloading workers that cache popular packages [109, 108].

5.3.4 Recent applicable solutions

As discussed in section 5.2, three recent algorithms deal with problems similar to ours [109, 108, 104]. While we could adapt these for the FaaS scheduling problem, slight differences in the problems they solve make them suboptimal alternatives.

First, lets consider the algorithms proposed by Nasir et al. [109, 108]. These algorithms have been designed to solve the problem of balancing the load of workers (processing element instances) in

a distributed stream processing engine like Apache Storm. These works make the strong assumption that splitting requests for a specific key between multiple workers is not desirable. In our case, it is acceptable to send tasks that require a package p to different workers; doing may not be optimal in terms of performance, but it does not affect the correctness of the system.

The problem tackled by Mirrokni et al. [104] is more similar to ours in that requests (balls) to a key k can be sent to different workers (bins) in the cases that the first target worker is overloaded—under a definition of overload that means: exceeding the average load by more than some ϵ. As they frame the problem as a classic *hashing* problem, their algorithm ensures that the assignments of balls to bins is deterministic (through the use of a linear probing technique). Our problem definition relaxes both assumptions. First, we tolerate high imbalance as long as individual workers are not overloaded. Second, if the affinity workers for an item are overloaded, we make no attempt to deterministically decide a fallback target, and revert to plain load balancing instead (by choosing the least loaded worker). Nevertheless, consistent hashing with bounded loads could potentially solve the problem outlined in this section. For this reason, we include this algorithm in our evaluations and empirically show that PASch yields better performance for our particular setting.

5.4 Proposed solution

We describe a solution to the problem formalized in section 5.3. Given a set of task execution requests and a set of worker nodes on which these can run, the goal is to implement a lightweight scheduler that maps requests to workers seeking to minimize the overall task turnaround time, while actively avoiding worker overload.

We propose PASch, a novel scheduling algorithm that leverages the power of 2 choices technique [105], consistent hashing [88, 138], and least loaded scheduling, adding the notion of a maximum per-worker load threshold. In this way, the restrictions of the FaaS scheduling problem are met.

PASch uses a greedy approach in which a mapping function $M : P \to W$ maps each request to two affinity workers for the *largest* package required by the task[9]. The least loaded of these workers will be the one to process the task, unless the load of the worker exceeds its execution threshold t_w; this provides affinity while avoiding worker overload. In addition, if the worker exceeds its threshold, then the scheduler reverts to a least loaded scheduler, and forwards the request to the worker with the (normalized) least load, for some specific resource unit. Algorithm 5 shows the details of the proposed procedure.

To provide stability, the mapping function $M : P \to W$ uses consistent hashing so that we minimize the expected number of movements in case worker nodes are added or removed by an auto-scaler or elasticity manager (see Algorithm 6). The use of consistent hashing ensures stability

[9]For simplicity, we do not deal with collisions of the affinity workers; in case of collision, there is only one affinity worker for a function.

Algorithm 5: Package-aware scheduler algorithm (PASch)

```
Global data: List of workers, W = {w_1, ..., w_n}, and their load thresholds, T = {t_1, ..., t_n},
             mapping function M
Input: Function, f, largest required package, p
1  if (p is not null) then
       /* Get affinity workers                                          */
2      ⟨a1, a2⟩ = M(p)
       /* Select target with least load                                 */
3      if (load(w_a1) < load(w_a2)) then
4          A := a1
5      else
6          A := a2
       /* If target is not overloaded, we are done                      */
7      if (load(w_A) < t_A) then
8          Assign f to w_A
9          return
   /* Balance load                                                      */
10 Assign f to least loaded worker, w_i
```

of the package-to-worker affinity mappings, even in the presence of a dynamic set of workers. We add a salt to the package id, p, to map it to a second affinity worker using the hashing function.

Algorithm 6: Mapping function, M; given a package, returns two affinity nodes.

```
Global data: A consistent hash implementation, consistent, and value to be added to the
             package ID to map a second affinity worker to it, salt
Input: Package id, p
Output: Affinity workers for p, ⟨a1, a2⟩
   /* Get two affinity workers                                          */
1  a1 = consistentHash.get(p)
2  a2 = consistentHash.get(p + salt)
3  return ⟨a1, a2⟩
```

5.4.1 Analysis

The performance of the scheduling decisions is dominated by how efficient it is to find the loaded worker. This operation is $O(n)$ if performed via a linear search—a frequently used approach in real implementations. For example, this is how the NGINX software load balancer implements its least-connected policy. Alternatively, we could use an efficient heap implementation. For example, using a Fibonacci heap [65], finding the least loaded worker takes $O(\log n)$ amortized time.

Regarding the balancing of the loads, we consider two situations. When the system load (sl) exceeds the overall capacity of the workers (i.e., $sl \geq \sum t_w$) the load is balanced according to a least

loaded policy and all the workers are similarly overloaded. When the system load does not exceed the overall capacity of the workers (i.e., $sl < \sum t_w$) the maximum load a worker w can have is t_w; however, some workers could have a load close to 0, in case the packages that have affinity to that node are unpopular.

5.4.2 Implementation in OpenLambda

We implemented PASch on `olscheduler`[10], a lean scheduler we developed for OpenLambda. We chose Go as the programming language as its primitives lets the developer easily build high-performant distributed systems. For the consistent hashing implementation, we used the *Package consistent* implementation [94], which uses the BLAKE2b collision-resistant cryptographic hash function [21]. The current implementation uses the number of active requests to measure the per worker load and the threshold is defined as a maximum number of active requests; this is a common approach used by load balancers. However, the algorithm works for other types of resource limits.

To facilitate the implementation of future schedulers, `olscheduler` exposes the following information:

1. Workers: Number and references to the workers.
2. Per-worker load: Measured as the number of active requests that each worker is currently handling.
3. Required packages: Each cloud function request includes the list of required packages, sorted by size.
4. Function schemas: Obtained by querying the code repository and cached in memory.

To get the number of required packages (item 3 above), we extended the HTTP Post request (cloud function request in Figure 2.1) so that it supports receiving the list as an annotation on the function call. An alternative design would have been to query the cloud store to get this information; we rejected this idea to avoid an extra step on the critical path of the requests.

`olscheduler` currently supports the following scheduling algorithms:

- Round robin: Distributes the requests uniformly between the workers, in round robin fashion.
- Least loaded: Assigns a request to the worker that has the least number of active connections (akin to NGINX's least-connected).
- Random: Distributes requests randomly between the workers, according to user defined worker weights.

[10] To seek early feedback, we presented `olscheduler` at the Min-Move workshop @ ASPLOS 2018, which has non-archival proceedings.

- PASch: The algorithm proposed in this chapter.
- Consistent hashing with bounded loads [104]: A recent variant of consistent hashing that provides a constant bound on the load of the maximum loaded worker.

To find the least loaded worker we use a linear search. For the implementation of consistent hashing with bounded loads, we use the *Package consistent* implementation [94], which has no configurable parameters and uses a balancing parameter of $c = 1.25$; this algorithm guarantees that the maximum load is at most $\lceil cm/n \rceil$, where m is the number of clients (current requests) and n is the number of servers (workers).

The implementation of PASch on `olscheduler` took only 52 lines of code.

5.5 Experimental evaluation

We assess the performance of PASch using simulations and a real deployment in a public cloud. The simulations run large-scale experiments with 1 000 workers. We also run real experiments on a public cloud, using a small deployment with 5 workers. Together, these experiments enable us to obtain a broad understanding of the benefits and costs of using PASch. Our experiments seek to answer the following questions:

Q1: How effective is PASch in increasing the import cache hit rate?

Q2: What is the cost (in load imbalance) of using PASch?

Q3: How much does PASch speed up individual tasks?

Q4: Is PASch successful in reducing median and tail task turnaround times?

Q5: How sensitive is PASch to the threshold parameter?

5.5.1 Experimental setup

We implemented a **simulator** in Python, using the SimPy simulation framework[11] and ran tests with 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 100 tasks simultaneously ($st = 100$).
- The popularity of the packages follows a Zipf distribution, with parameter $s = 1.1$.
- The time to start the packages—time to download, install and import—is randomly sampled from an exponential distribution with an average time to start of 4.007s.

[11] `https://pythonhosted.org/SimPy/`

- 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 500MB LRU import cache.
- 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 of all nodes: $t = st = 100$.

While the configuration described above represents an artificial scenario, the values model real observed behavior, as reported by prior work [79, 111, 113, 97, 3]. To isolate the effect of PASch on the import cache, the simulations do not use the install or handler caches. The simulator supports three scheduling algorithms: (1) least loaded, which optimizes for load balancing, (2) hash affinity, which optimizes for cache affinity, and (3) PASch, which combines both goals as described in this chapter.

We also ran **real experiments** on AWS EC2. We use six virtual machines (VMs) with the 4.4.0-1072 Linux kernel. On one VM we run `olscheduler` and PipBench; on the other five ones, we launch five OpenLambda workers. We used m4.xlarge instances, with 4 vCPUs, 16 GB of RAM, and EBS storage. The experiments used OpenLambda's lightweight SOCK [113] containers. In the real experiments we used `olscheduler` which, as described in section 5.4.2, supports five scheduling algorithms: three algorithms that are common in software load balancers like NGINX (round robin, least loaded and random) and two algorithms that consider scheduling affinity in addition to load management (PASch and consistent hashing with bounded loads). For the case of PASch, we configure the threshold parameter to 80. The consistent hashing with bounded loads implementation we are using (*Package consistent*) has no configurable parameters and uses a balancing parameter of $c = 1.25$, as recommended by Mirrokni et al. and an industry blog [127, 104].

All three caches were active during the experiments: the on-disk install cache, a 6GB import (package) cache and a 1GB handler cache. While PASch seeks to improve the import cache hit rate, the hit rates of the other caches also increase as a result of better package affinity. The install cache also deals with packages, so affinity to the import cache also yields affinity to the install cache. The handler on-demand cache stores function handlers at each worker; affinity to the largest package required by a function would mean subsequent calls to the same function would also end up going to the same worker. For a study of what percentage of the performance improvement comes from

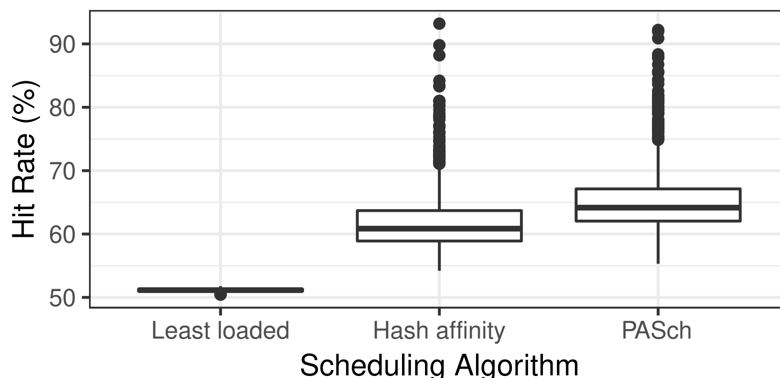

Figure 5.3: Box plots of the hit rates of the import caches at the workers. PASch improves the average hit rate by actively seeking to improve package-affinity.

using each of the caches, we refer the reader to the work of Oakes et al. [111, 113], who studied the performance of the caches on OpenLambda, using vanilla NGINX (least-connected).

We use the PipBench benchmark [111] to issue requests to function handlers which import packages from a repository populated with packages generated to emulate the directory structure, file sizes, dependencies, and popularity from real packages hosted in the web. We used the default configuration of PipBench.

5.5.2 Experimental results

Q1: Effect on import cache hit rate

The main premise of our solution is that increasing package affinity during scheduling decisions can improve performance as a direct result of an increased import cache hit rate, even at the cost of a manageable node imbalance. To confirm that we can improve the cache hit rate, we ran simulations to assess the impact that PASch has on the import cache hit rate. The results show that the median hit rate increases from 51.2% with the vanilla least loaded scheduler to 64.1% with PASch (see Figure 5.3). Note that the median hit rate with PASch is slightly higher than with the hash affinity scheduler; this is due to the finite size of the cache. Without a per-worker request limit, the resulting working set may not fit in the cache.

Q2: Effect on load balance

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 ran simulations and count the total number of tasks assigned to each worker, and show the coefficient of variation for every 1-second timeslot in the experiment in Figure 5.4. We

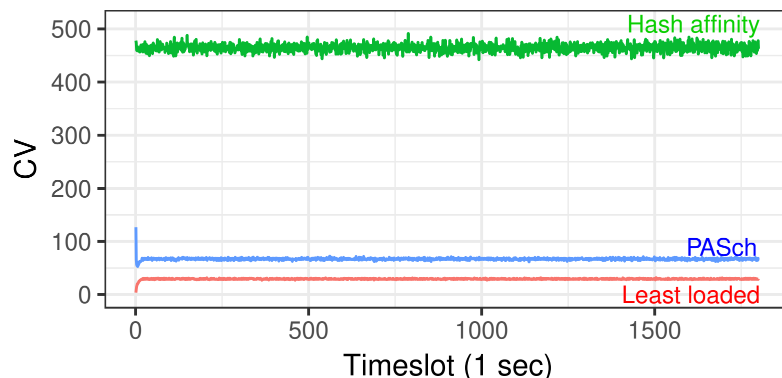

Figure 5.4: Node imbalance: Coefficient of variation of the number of tasks assigned to each worker during each 1-second timeslot (smaller is better).

can observe that PASch sacrifices some node balance, to seek a higher hit rate (and smaller task turnaround time). Least loaded achieves near perfect balancing, while the hash-based affinity algorithm produces the most unbalanced task assignments. We show next that the use of the threshold parameter in PASch is able to effectively contain the imbalance to a level manageable by the worker nodes, and as a result, the task turnaround times improve, even in the presence of (moderately) unbalanced worker loads.

Q3: Effect on individual tasks

We ran real experiments to observe the effect of PASch on individual tasks. To do this, we first calculate a performance baseline for each task by executing it on an empty cluster so that the task can finish as fast as possible, with no restrictions on resources at the worker, but without the benefit of caching. After having the performance baseline for each task, we used PipBench to asses if each task runs slower or faster than its baseline. For each task, we calculate the speedup. A task's speedup may be greater than one if, for example, the task executes faster through the benefit of better cache affinity. On the other hand, a task's speedup may be less than one if it runs on a node where other tasks are running and the concurrent tasks are competing for the use of the CPU. Figure 5.5 shows that PASch is able to improve the performance of most tasks through cache affinity, at the penalty of some tasks performing slower due to imperfect load balancing. Specifically, only 3.3% of the tasks have a speedup of less than one, while the median speedup is 3841. These speedups represent the maximum achievable speedups for the tasks when compared to an unoptimized baseline (no caching) but without incurring in performance penalties due to worker overload.

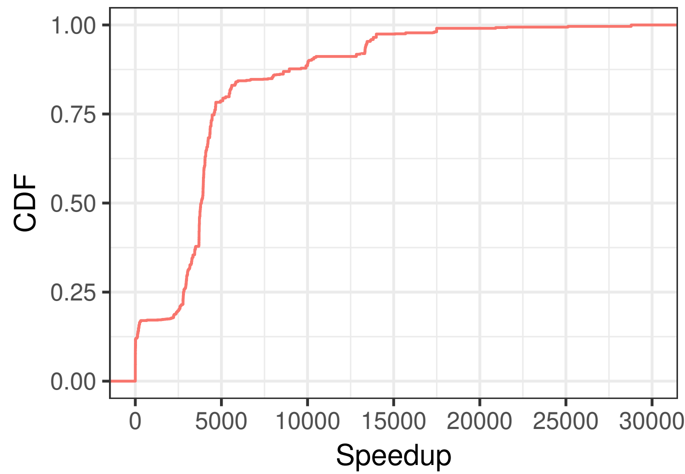

Figure 5.5: CDF of per-task speedups with PASch versus each task running on an empty cluster, with all resources available but no pre-cached packages.

Q4: Effect on median and tail task turnaround time

Having confirmed that PASch is able to improve the hit rate of the import caches of the worker nodes and that the worker imbalance is not so excessive as to decrease task performance, we assess the effect of PASch on task turnaround time (latency), when compared to other scheduling algorithms. Towards this end, we ran real experiments with PipBench. Figure 5.6 shows the cumulative distribution functions (CDFs) of the task turnaround times, when using different scheduling algorithms. On average, PASch achieves a 1.29x speedup compared to the least loaded scheduler. However, if you analyze the CDFs, you'll notice that the improvement is most noticeable in the $[61 - 87]$ percentile range. For example, the 80th percentile in task turnaround time has a 23x speedup. Consistent hashing with bounded loads performs better than least loaded, but PASch is still outperforms it for the workload studied in this chapter: PASch achieves a 1.3x average speedup, and speedups of 1.5x, 2.7x and 1.04x at the 80th, 90th and 99th percentiles, respectively.

Q5: Sensitivity to threshold parameter

PASch has one configurable parameter: the *threshold* value. Our model assumes that a worker has limited capacity and a known execution threshold t. To assess the sensitivity to this parameter, we run real experiments and evaluate the resulting task turnaround time, for varying values of t. Figure 5.7 shows that PASch is not particularly sensitive to the threshold parameter.

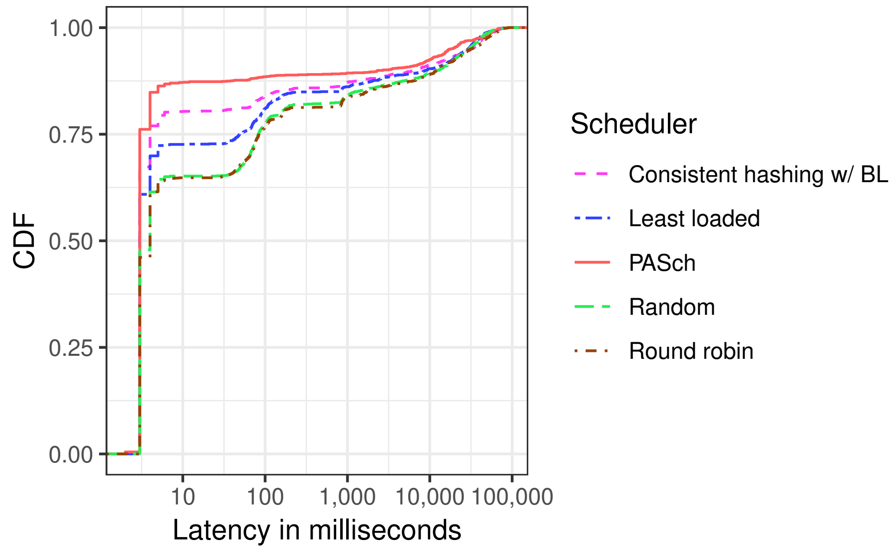

Figure 5.6: CDFs of the task turnaround times, when using different scheduling algorithms. PASch outperforms all other scheduling algorithms we evaluated.

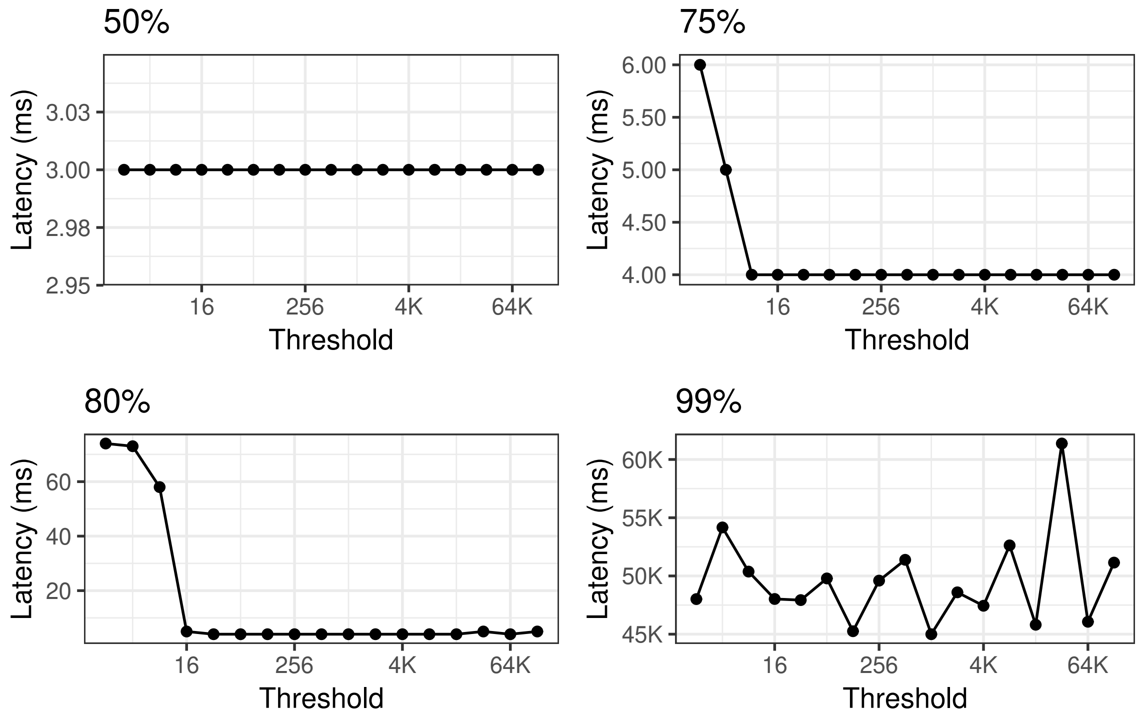

Figure 5.7: Effect of threshold on 50th, 75th, 80th and 99th percentile latency.

5.6 Conclusions

The main reason for load balancing is to improve performance, as tasks assigned to overloaded workers are bound to be delayed in their completion. However, the moderate imbalance of our proposed algorithm is not an issue, as our experiments show that we actually improve performance: tasks that run on workers that have preloaded a required package, take less time to finish; by improving the cache hit rate, we improve overall system performance.

Two key insights from our work are: (1) near-data scheduling techniques for FaaS platforms can yield significant performance improvements, yet all the open source FaaS platforms that we've looked into use vanilla load balancers; and, (2) when redirecting requests to workers, balancing the load of workers may not be a necessary goal: our experiments showed that relaxing this requirement and changing it to "avoid worker overload" gives us more flexibility in mapping decisions that can lead to considerable performance gains.

We have released the code of our scheduler on GitHub[12].

[12]https://github.com/disel-espol/olscheduler

Chapter 6

Performance-aware deployment of containers

Cloud-native applications are increasingly adopting microservices architectures that support the development agility required by modern software. These applications deploy their components in containers that enable microservices to be deployed across different platforms, supporting the independent scaling of the different components. However, the operational complexity of microservices presents significant challenges in maintaining the performance of such applications, especially in clouds with performance variability and unpredictability. While virtual machine based deployment of applications has been well studied—with sophisticated orchestrators in the literature and practice—there has been little such studies on the opportunity in improving application performance using performance-aware deployment strategies for containers. In this chapter, we consider both the run-time and initialization time performance of containerized applications and show that default placement strategies provided by orchestrators are often insufficient. Our experiments on multiple services show that a performance-aware approach is able to outperform the default placement strategy by up to factor of 2x and 2.21x for the 50th and 99th percentiles.

The work presented in this chapter was first published at the 5th International Workshop on Container Technologies and Container Clouds (WOC '19) [32].

6.1 Introduction

Microservices architectures have become common in modern software development. In a 2018 survey, 91% of developers at enterprises with more than 500 employees reported that they are using or have plans to use microservices in the near future [125]. However, adopters report problems meeting performance goals and solving performance issues, and—as a solution—74% of adopters plan to

increase investment in microservice performance management [125].

Under a microservices architecture, applications deploy their components in containers managed through container orchestration systems. This enables scaling the different components independently, provides benefits to the development process, and facilitates the attainment of important non-functional requirements like resiliency and fault isolation. However, the operational complexity of containerized applications presents significant challenges in maintaining the performance of such applications, especially in cloud environments with performance variability and unpredictability. While optimized initial placement of containers can help meet application service level objectives (SLOs) to a certain extent, workload spikes and poor container isolation [100, 164, 89] often result in performance degradation over the lifetime of the container. Further, such an approach would require resource reservations for the peak workload resulting in poor infrastructure utilization.

Adapting to performance variability in traditional architectures—where applications are deployed on virtual machines (VMs)—has been well studied, with sophisticated solutions that perform mitigating actions like creating new replicas or VM migration, which are typically slow and expensive operations [115, 152]. In contrast, containerized applications are more lightweight and designed to be more motile when compared to VMs, making it feasible for the containers to be scaled out or migrated faster than VMs. Container orchestrators like Kubernetes (k8s) and Swarm provide scale-out mechanisms as a principal feature to maintain the application SLOs during workload spikes and infrastructure performance episodes.[1] However, we find that the default scale-out deployment mechanisms offered by the platforms (as described in section 6.2) are not performance-aware, resulting in poor application performance.

Container placement has recently gained some research interest. DRAPS [100] maps containers to nodes in heterogeneous clusters based on required resources. Have et al. [76] learn consumption profiles of containerized applications and use them in placement decisions. Chung et al. [44] make placement decisions seeking to reduce the monetary cost of containerized batch tasks. We complement these works by studying how container placement decisions affect performance goals. We find that even when a node has available resources, the lack of performance isolation has an impact on the launch time and steady phase performance. This implies that bin packing algorithms are not sufficient to meet latency requirements in performance-sensitive deployments. Our work considers goals relevant to elastic services and not those pertinent to batch jobs or data processing short-lived tasks (that are applicable to applications running on data processing platforms like Spark [81]).

We study the benefits of performance-aware deployment of containers in improving application performance across two dimensions: (i) *run-time container performance*, which affects the performance of the application running in the container, and is critical for meeting application SLOs, and (ii) *container initialization times*, which affects how quickly a service can respond to the environment dynamics. The latter is particularly important for ephemeral containers that are typically created

[1] K8s has support for vertical auto-scaling which can be combined with its scale-out mechanisms; we did not study this feature to analyze horizontal scaling in isolation.

to handle workload spikes.

The main contributions of our work are as follows. First, we study the impact of available CPU on the host nodes, on the application latency at individual containers. Based on this, we show the importance of deploying the containers on nodes with sufficient CPU resources to meet the application SLOs, and make a case for more work on performance-aware deployment of containers. Second, we present the design of a closed-loop system that monitors the container's performance and performs scale-out on the right set of nodes. This helps sustain the required application latency by adapting to the infrastructure dynamics, and can lead to latency improvements of 2x or more. Finally, we present some of the principal challenges in building the system based on our observations.

6.2 Background and motivation

Datacenters that support cloud platforms with dynamically-requested resources need mechanisms for on-demand co-location of workloads in the same physical machines while meeting tenant service level objectives (SLOs). This is the **workload placement** problem of mapping virtual resources to physical resources and corresponding realization in the datacenter infrastructure [18]. This problem has been studied in the context of multi-tier web application placement [140], virtual machine (VM) placement [153, 18, 25], application placement [53] and placement of jobs composed of tasks [152, 81].

Each case of the workload placement problem deals with its own requirements and restrictions, given by the infrastructure and workload characteristics. **VM placement** for clouds differs from the problem studied in this chapter in three important ways: (1) Works in this domain typically seek to minimize consumed resources (and energy) by tightly packing VMs in physical machines (PMs), e.g., as in [30]; (2) VMs take longer to launch, so improving their launch time through placement decisions is not applicable; and, (3) VMs have stronger isolation properties and clear resource requirements (as given by the type of VM being deployed), which facilitates good placement decisions. In contrast, for the case of services running on containers on a cluster of VMs, the amount of resources is fixed (and controlled by the VM placement layer); smart placement can lead to improved launch and service times; and, the limited isolation complicates the task of meeting tenant SLOs.

Early work on (Linux) **container scheduling**—done for the context of distributed processing platforms like Apache Spark—looked into the case of a very large system in which the scheduler must be distributed [81] and also sought to provide support for very short-lived tasks that require scheduling decisions to be extremely fast [114]. Furthermore, the main performance goal applicable to these platforms is job turnaround time, and not request service time as in the problem studied in this chapter. We study the most common use case for container orchestrators like Kubernetes and Docker Swarm: deploying multi-tier or microservice-based applications, for which massive scheduling and short-lived tasks are not an issue.

In sum, mapping containers to VMs in platforms supporting service-based applications differs

from prior research in workload placement in the assumptions being made about the workloads and the infrastructure. We look into goals relevant to elastic services instead of batch jobs or data processing short-lived tasks; where the end goals are not minimizing power consumption, maximizing resource usage, or reducing job completion times, but rather improving the performance of the services running in these containers—in terms of initialization times, or steady state request response times.

Next, we describe the placement algorithms implemented by the two container orchestration frameworks commonly used in current service-based applications: Docker Swarm[2] and Kubernetes [37].

Swarm supports three placement algorithms[3]: spread, binpack and random. Spread selects the node with the least number of containers. Binpack selects the node which is most packed.

We focus on Kubernetes, which supports placement restrictions based on affinity and anti-affinity, resource requirements, and user defined labels. This information is used to filter nodes during a first phase. In the second phase, the scheduling algorithm ranks nodes according to priority policies with the following goals[4]:

- Spreading service to improve reliability (SelectorSpreadPriority, ServiceSpreadingPriority).
- Preferring or avoiding nodes based on user-provided rules (InterPodAffinityPriority, NodePreferAvoidPodsPriority, NodeAffinityPriority, TaintTolerationPriority, CalculateAntiAffinityPriority).
- Seeking to balance load or CPU/memory consumption (BalancedResourceAllocation, LeastRequestedPriority).
- Bin packing (MostRequestedPriority, RequestedToCapacityRatioPriority).
- No special priority (default): Round robin (EqualPriorityMap).
- Reduce dependency load time (ImageLocalityPriority).

While these algorithms can be used to meet goals related to resource usage, load balancing and reliability, only one of them considers performance (ImageLocalityPriority), and none is suitable for applications that seek to meet tight performance SLOs.[5]

6.3 Design

Microservice applications are deployed within containers that are hosted on physical or virtual host nodes. These host nodes are often grouped together into a cluster, which is managed by a container

[2] `https://docs.docker.com/engine/swarm/`
[3] `https://docs.docker.com/swarm/reference/manage/`
[4] `https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/`
[5] Parallel to this work, IBM released SSX, a scheduler expansion for k8s that avoids overloading the workers, considering actual resource usage instead of just the amount of allocated resources. We discuss SSX in section 6.5.

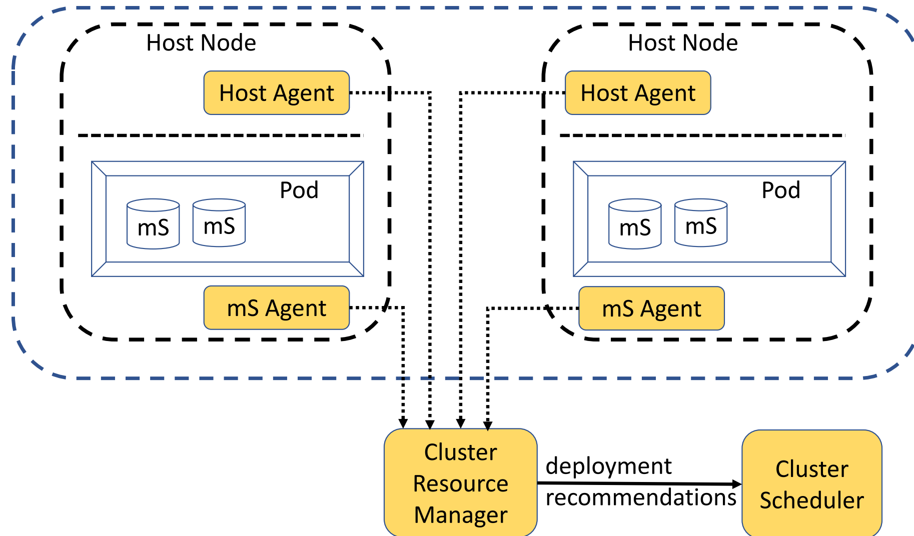

Figure 6.1: Proposed design.

management platform. Application microservices are typically grouped into pods, which represents a unit of deployment consisting of containers that are tightly coupled and share resources.

Figure 6.1 shows the design of a performance-aware system for container deployment. There are three key components to our design: (i) Host monitoring Agents that monitor and collect the resource availability at the hosts; in this chapter, we monitor CPU resources using `vmstat`[6]. (ii) Microservices Agents that monitor the application performance and detect any SLO violations; these could be profilers that are integrated with the application or health-check containers that periodically ensure the liveness of the microservice. Furthermore, the agents can leverage metrics captured for DevOps, e.g., as reported to Prometheus[7]. (iii) A cluster resource manager that receives the resource availability from the Host Agents and performance metrics from the Microservices Agents, and provides deployment recommendations to the k8s scheduler.

We consider the run time and the initialization time performance of the containers to determine the deployment strategy. The run-time container performance determines the ability of a microservice to sustain the required level of performance; the initialization times of a container is key to meeting SLOs, especially when responding to sudden and unexpected spikes in workload. Based on our initial study, we find it is important to measure the microservice performance as a function of available host resources, when making scale-out deployment decisions for containers. The Resource Manager (RM) acts as a closed loop system and evaluates the impact of available host resources (measured through the host agents) on the microservice performance (observed through the microservices agent). Based on these measurements, the placement and scheduling algorithms in the

[6] `https://linux.die.net/man/8/vmstat`
[7] `https://prometheus.io/docs/introduction/overview/`

Table 6.1: Summary of the experiments presented in this chapter.

No.	Service	Research Question	Section	Objective
E1	TeaStore Auth	Q1	6.4.1	Study impact of performance-aware scheduling in service response times.
E2	`textRotate.jspx`	Q1	6.4.1	Study impact of performance-aware scheduling in service response times.
E3	NGINX	Q2	6.4.2	Study impact of performance-aware scheduling in service launch time.

RM recommends an optimal deployment strategy to the Cluster Scheduler (e.g, `kube-scheduler`).

6.4 Evaluation

We present results that show how CPU consumption at the workers affects service performance, hindering an application's ability to meet its SLOs; and, how smart placement decisions can significantly improve performance. We studied three CPU-intensive services: (1) An authentication microservice from the TeaStore microservices reference application [154], (2) the `textRotate.jspx` service that comes with Apache Tomcat[8], and (3) the NGINX[9] reverse proxy, commonly placed in front of the microservices layer to forward REST requests to the proper microservice. Each service ran in a Docker[10] container and the system is orchestrated using Kubernetes[11]. The services were deployed one per node, to avoid performance interference. CPU stress was added to the nodes using `stress-ng`[12]. All experiments were performed multiple times to ensure the results were repeatable. In this section, we provide aggregate results or results from a single representative run, as indicated in each experiment. The experiments presented in this section were designed to answer the following questions (see Table 6.1):

Q1: Can performance-aware scheduling lead to improved response times (for containerized services)?

Q2: Can performance-aware scheduling lead to reduced (containerized) service launch time?

Setup: We used CloudLab [56] machines with Ubuntu 18.04.1 LTS, K8s 1.13.3, and Docker 18.09.7. For the service response times experiments, we used `d430` and `r320` bare metal nodes (for the TeaStore and textRotate experiments, respectively). These were selected as they are representative of what is available for lease at cloud providers. For the container initialization experiments we used `pc3000` nodes, as the single core eased in partially stressing the CPU. For validation, we repeated the experiments on `d710` nodes and ensured that the results hold for different hardware configurations. For detailed hardware specs, see the CloudLab documentation[13].

[8] `http://tomcat.apache.org/`
[9] `https://nginx.org/en/`
[10] `https://www.docker.com/`
[11] `https://kubernetes.io/`
[12] `https://kernel.ubuntu.com/~cking/stress-ng/`
[13] See: `http://docs.cloudlab.us/cloudlab-manual.html`

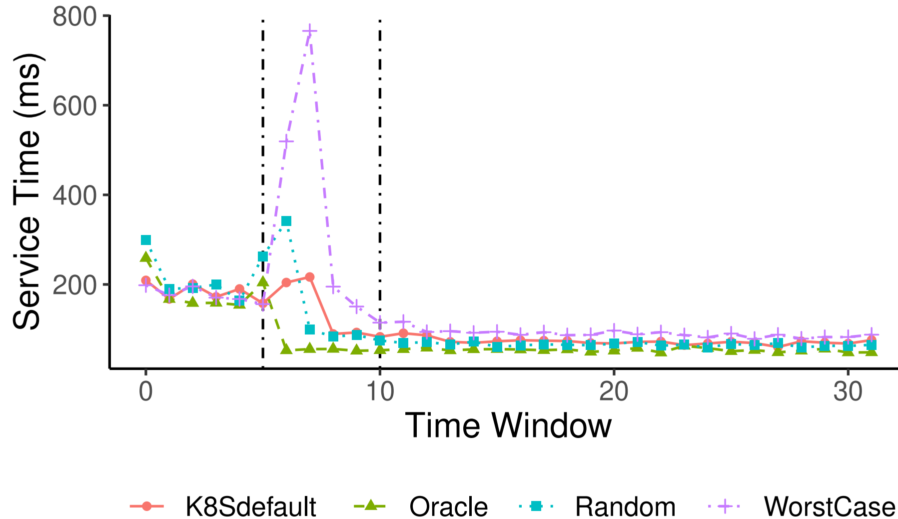

Figure 6.2: Average response times for the TeaStore auth microservice, for every 10-second window of the experiment. The vertical dashed lines separate the three phases of the experiment: base (1 replica of service), transition (a new replica is launched), and steady (2 replicas of service).

Schedulers used in the experiments: K8Sdefault, Random, WorstCase and Oracle. The Oracle and WorstCase schedulers choose the node with the most or least CPU load, respectively. Oracle uses perfect system knowledge to give an upper bound on the improvements that we could achieve through smart placements decisions.

6.4.1 Response times: Experimental design and results

To study how scheduler placement decisions affect service response times, we performed experiments in which initially there is only one instance of each service, running on an independent container; we call this the *base phase*. We do not show results of the performance during the base phase, as they are the same for all the schedulers since they all begin by placing the original service on an empty worker node. At this point, we add 50% CPU stress to one node, and as a result the Kubernetes cluster now has two empty worker nodes (with and without CPU stress) and one worker node for each containerized service. We then launch a new replica of the service being studied. In all the cases, we observe decreased performance (spike) during the time the new service is being brought up; we call this the *transition phase*. After the performance spike subsides, the system reaches a *steady phase* during which the response times are affected by the scheduler placement decision, with decreased performance observed at the node with CPU stress. Figure 6.2 illustrates these three phases, for one run of the TeaStore experiment.

Our main interest is in the performance during the steady phase, but we also include results obtained during the transition phase. We identify this phase by studying the time series of the

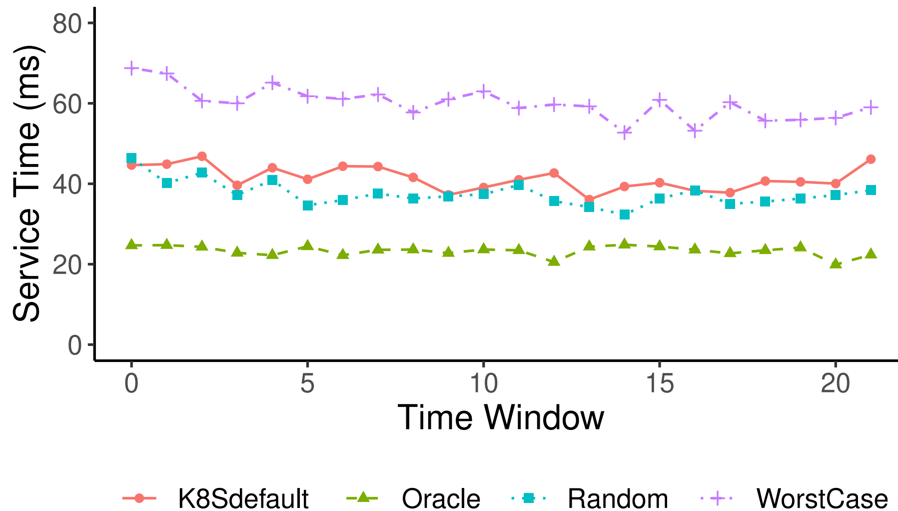

Figure 6.3: Median response times for the TeaStore authentication microservice, for every 10-second window of the steady phase of the experiment.

requests and selecting the results after the performance degradation (spike) that occurs right after deploying an additional service replica.

We issued requests as described next. (E1) We ran the TeaStore application with one microservice per node, and issued requests with the LIMBO HTTP Load Generator[14], with a profile that simulates a user browsing the site and adding products to the shopping cart. (E2) For the `textRotate.jspx` tests we used ApacheBench[15]. The requests issued had a random string in the service call, to avoid the effect of result caching which is enabled by default in Tomcat.

E1 results: Figure 6.3 shows the median response time at the steady phase, for TeaStore's authentication service. The results show that the Oracle scheduler can lead to median performance that is, on average, 1.75 times better than the one resulting from the Kubernetes scheduler. At the 90th and 99th percentiles, Oracle is 1.25x and 1.17x better than K8Sdefault. We observe that random decisions can lead to better performance results than Kubernetes. Response times degrade more during the transition phase—due to inadequate placement decisions—as shown in Figure 6.4: Oracle leads to response times that are 2x, 1.77x and 2.21x better than those obtained with K8Sdefault, for the 50th, 90th and 99th percentiles, respectively.

E2 results: Figure 6.5 shows the cumulative distribution functions (CDFs) of the response times. As in the prior example, Oracle leads to better service response times than K8Sdefault, yielding service response times that are 1.5x, 1.43x, and 1.13x better than those obtained with the Kubernetes scheduler, for the 50th, 90th and 99th percentiles of the steady phase. During the transition phase (not shown), the service response times obtained with Oracle are 1.11x, 1.48x, and 2.06x better than

[14] `https://github.com/joakimkistowski/HTTP-Load-Generator`; model detailed in [155].
[15] `https://httpd.apache.org/docs/2.4/programs/ab.html`

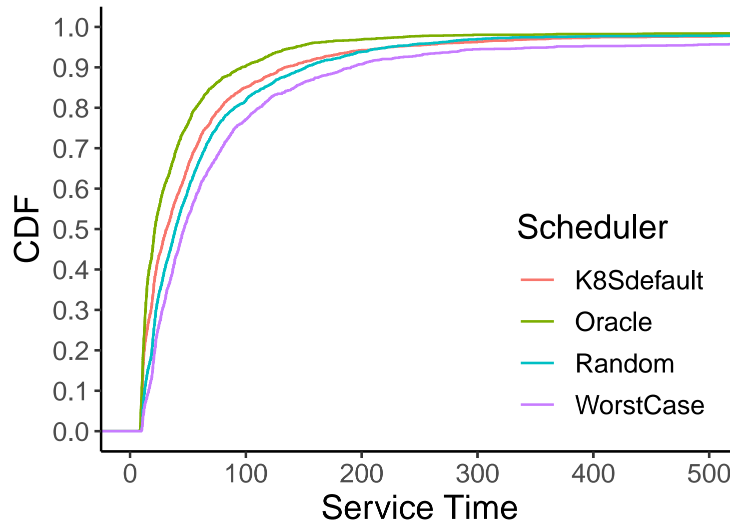

Figure 6.4: Cumulative distribution functions (CDFs) of the service response times during the transition phase (time during which a new replica is being launched), for the TeaStore authentication microservice.

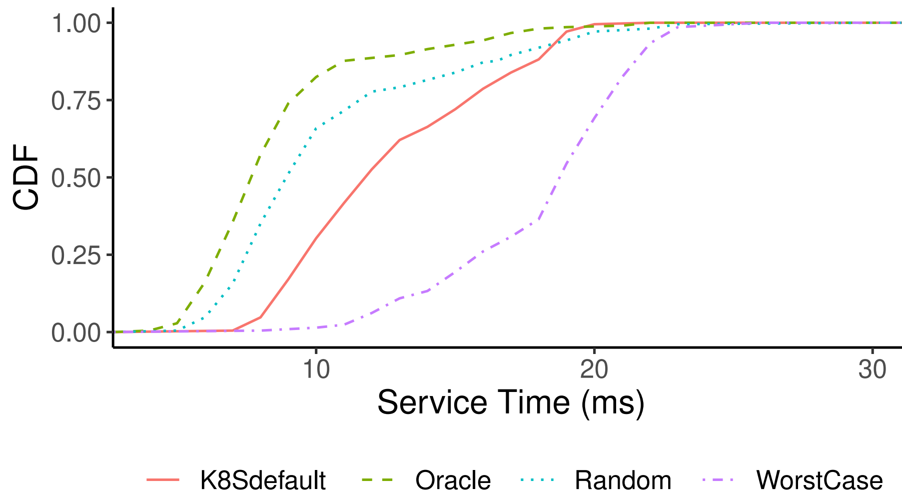

Figure 6.5: CDFs of the service response times of the Tomcat textRotate.jspx service, during the steady phase of the experiment.

those obtained with the Kubernetes scheduler, for the 50th, 90th and 99th percentiles, respectively.

Insights: Smart placement can lead to significant improvements in median and tail service response times, during the transition and steady phases of the service replica creation process. The default placement decisions of the Kubernetes container orchestration platform lead to sub-optimal performance. Our observations call for improved, performance-aware placement algorithms that target the performance metrics relevant to modern containerized services.

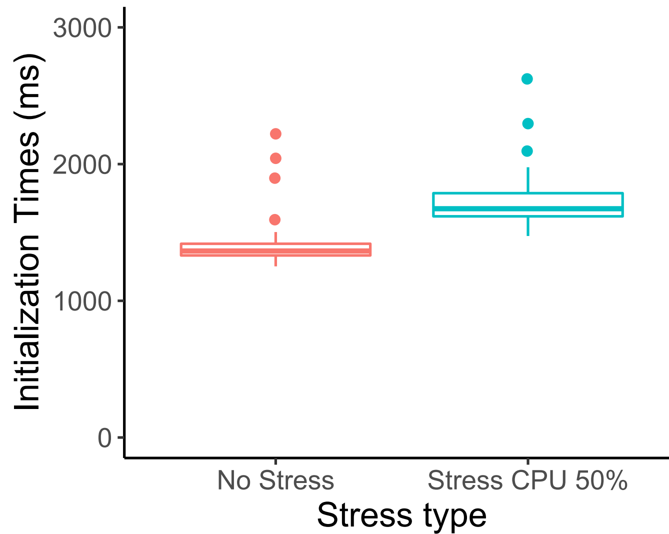

Figure 6.6: Box plots of the container initialization times of 50 iterations of an experiment in which an NGINX instance on a Docker container is launched on an empty worker node or on a node with 50% CPU stress.

6.4.2 Initialization times: Experimental design and results

Some services require fast service launch times in addition to fast service response times. This is of importance to: (a) Elastic services that auto-scale in response to frequent variations in workload demand [80], and (b) Function-as-a-Service, for which slow cold-start times is known to be an issue hindering adoption [149].

E3 results: To assess the impact of CPU load on how fast a service launches, we devised a microbenchmark in which we launch NGINX on an empty node and on a node with 50% CPU stress. We ran this test 50 times and show the distribution of the initialization times in Figure 6.6. Launching the service in the node with no stress versus launching it on a node with 50% CPU stress is 1.23x, 1.34x and 2.19x faster at the 50th, 90th and 99th percentiles, respectively.

We also report that we were originally running the experiments on Ubuntu 16.04 and found that the performance degraded significantly for every iteration in the experiment (see Figure 6.7). After extensive testing, we were able to track down the issue to a kernel bug that lead to cgroups not being properly released[16]. A fix is included in newer versions of Ubuntu, but many production systems are still running the problematic version.

Insights: Launching a containerized service on a node with (even medium) CPU stress can lead to reduced service initialization times. For services that require to launch fast, monitoring and smart placement decisions become important. Furthermore, continuous feedback loops are important to detect possibly unknown performance issues, as with the case of the cgroups bug that considerably

[16]See: `https://github.com/lxc/lxc/issues/1443`

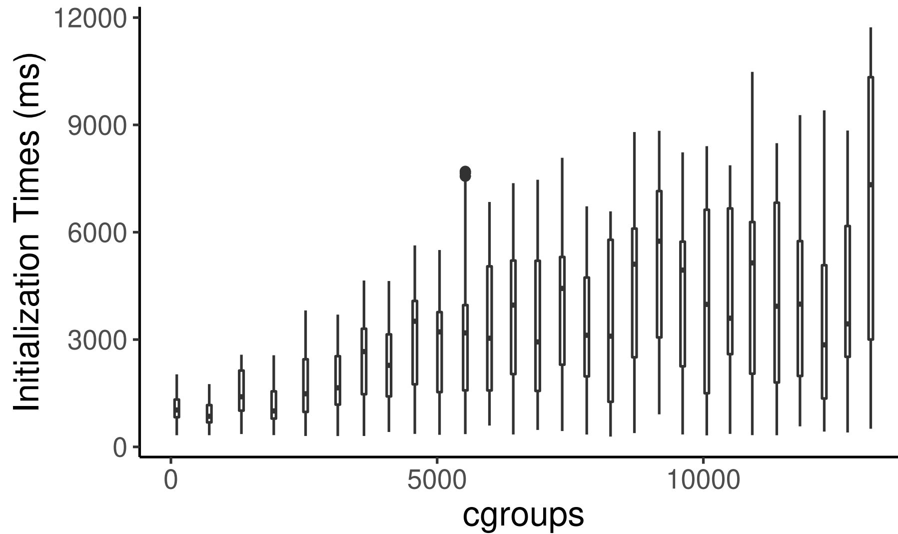

Figure 6.7: Performance degradation of the container initialization times, as the number of cgroups in the system increases as a result of a kernel bug.

affects container performance in nodes with high container churn.

6.5 Related work

For related work studying the problem of workload placement in diverse scenarios, we refer the reader to section 6.2.

There have been a few recent efforts seeking to improve **performance isolation** between containers; e.g., [164, 89]. Others are working on alternatives to VMs and containers that mix the best of both worlds: lightweight application deployment with strong isolation; namely, research prototypes like Alto [95], Cntr [141] and X-Containers [132], and commercial offerings like AWS Firecracker[17]. These approaches are orthogonal to our work, as the performance impact perceived by microservices can occur due other issues like normal performance degradation at nodes with medium-to-high resource consumption levels, and bugs at lower layers.

Within the context of **container placement**, Mao et al. [100] analyzed the Docker Swarm orchestrator and found that it did not consider the node resources or container resource requirements when mapping containers to nodes, but rather applied a simple *spread* container algorithm. They proposed DRAPS, which assigns containers to nodes based on current available and dynamic demands from the services running on the containers. Our work complements this work, by studying a more complete container orchestration engine: Kubernetes, and exposing the fact that even when a node has available resources, the lack of isolation in the performance on containers has an impact

[17] `https://firecracker-microvm.github.io/`

on the container launch time and steady phase container performance, which calls for performance-aware algorithms that consider not only resource consumption but also application-level performance metrics. Have et al. [76] proposed making Docker Swarm more energy efficient. In contrast, we seek to improve the performance of containerized applications with tight SLOs. Chung et al. proposed Stratus [44], a scheduler for batch tasks running on containers (on a public IaaS) that seeks to minimize the monetary cost of running the tasks. In contrast, we look at performance improvement given a fixed set of resources.

Specific to **Kubernetes**, Medel et al. [102] derived a reference model for Pod and container lifecycle management. Podolskiy et al. [118] looked into how to trigger container auto-scaling decisions based on application performance models and user defined SLOs. Truyen et al. [146] studied the performance overhead of container orchestration frameworks for management of multi-tenant database deployments. Our work adds another dimension into the studies of the performance of applications running on Kubernetes.

The wide adoption of **microservices** architectures and their performance challenges has lead to work in benchmarking [67][18], performance debugging [68], predicting tail latency [123] and SLO violations [86], self-healing and self-adjusting orchestrators [87], auto-scaling [26], and performance on serverless platforms [97]. Our work complements these projects, by studying how the launch and service times of microservices is affected by deployment decisions.

Recently this year, IBM released SSX[19], a scheduler expansion for k8s that avoids overloading the workers, considering actual resource usage instead of just the amount of allocated resources. Our evaluation results provide experimental evidence that this direction of work is promising. SSX could be used as a basis for future work on smart performance-aware deployment of containers.

6.6 Closing discussion

Lowering the latency of modern applications is critical for user engagement and profits [135]. In this chapter, we highlight the opportunities to improve application performance through performance-aware deployment of containers. We showed that the default scale-out deployment mechanisms offered by popular container orchestration platforms are not performance-aware, resulting in poor performance of the microservices. In particular, we find that placement decisions are critical for both good initialization time and run-time performance of the containers—metrics that are essential for containerized applications with strict latency SLOs. Further, our observations highlight the need and benefits of a closed-loop system that helps sustain the application SLOs by scaling or migrating containers based on the resource variability in cloud environments. We now present some key considerations for building such a system.

[18] At the time of this submission, DeathStarBench has yet to be released.
[19] `https://github.com/IBM/kube-safe-scheduler`

Smaller time scales: Modern cloud-native applications are designed to be fast and are often required to maintain response times of a few milliseconds [87, 68, 123]. Containerized applications are significantly lightweight when compared to their VM counterparts, which presents the opportunity for these applications to scale-out/scale-in quickly in reaction to performance variability in the cloud platforms. However, this requires the placement algorithms in the container orchestration platforms to match these strict latency requirements. Further, our observations show that it is important to ensure the stability of the closed-loop system, as oscillations could create significant deployment churn in the cluster, which drastically impacts the application SLOs as we show in section 6.4.

Deployment complexity: Containerized applications differ from applications designed for deployment in traditional VMs, primarily due to the fine-grained functional decomposition of the application into microservices that are deployed in containers. End-user transactions typically traverse chains of microservices, increasing the length of the critical path impacting the applications' end-to-end latency. Hence, localized placement decisions which are better for individual containers, may differently affect the end-to-end latency. It is important for the placement algorithms to retain a holistic view of application performance when (re)deploying containers.

Accommodating application constraints: Applications often specify constraints that affect the placement of its individual containers; e.g., require a set of containers to be co-located for performance. Also, microservices may need containers to be placed on hosts with specific hardware features. Placement algorithms need to be cognizant of these constraints while making (re)deployment decisions.

Horizontal scaling vs. vertical scaling vs. migration: Prior studies [126, 77] have shown that application scale-out may not always be the right action to mitigate poor performance. For the case of containerized applications, this may be especially relevant for applications with complex interactions across microservices where root-cause analysis is not straight forward. Thus, in many cases, vertical scaling or migrating to a new host may be more appropriate.

Analysis period: Our experiments had constant CPU stress, but real applications consume resources in variable quantities during their executions. The resource consumption at workers should be monitored during some period in order to make good placement decisions. This period could be longer when we want to improve request response times, but shorter when our goal is to reduce initialization times. As applications may care about both metrics, properly choosing the period to analyze prior to a placement decision is a challenge for smart resource-aware placement algorithms. These observations call for more research into performance-aware placement algorithms designed for containers managed by orchestrators, running on clusters of VMs managed by a VM placement platform. Researchers could also develop auto-configuration mechanisms that periodically tune the orchestrator's configuration knobs, such as to maximize performance. Recent work in the domain of self-driving databases [147] has showed that such approaches are useful and can help in reducing operational costs while improving system performance.

Chapter 7

Cache affinity request routing with load control (CARLOs) for data lakes

Big Data processing has become a popular applications that makes use of cloud platforms to optimize resources and reduce costs, and requires large storage capacities. Data lakes are large, highly scalable storage systems that have emerged to optimize the performance of analytical tools such as Hadoop or Spark. Data lakes require fast access, and use common techniques like caching and prefetching to reduce response times to data. In this chapter we propose the use of CARLOs, an algorithm that seeks to maximize the use of the cache (hit rates), without neglecting load balancing, as a method to obtain better access times to the data stored in the data lake. Our experiments shows that CARLOs outperforms the standard request routing algorithm, reducing request latency by up to 20%, and increasing the throughput up to 9%.

7.1 Introduction

To demonstrate that our affinity scheduling approach for serverless microservices has important applicability (impact) beyond the microservices domain, we now turn our focus into a current important problem in cloud computing: Making Big Data processing more efficient. Big Data processing is one of the fastest growing modern applications that makes use of cloud platforms in order to optimize resources and reduce costs. Massive data processing has the characteristic that it requires large storage capacities, at least temporarily, to make the data available to the processing tools.

Massive data processing is done through the use of frameworks, such as Hadoop or Spark, which can work with data flowing directly from the original sources, transformed and processed by these

frameworks, or with data that has been previously processed, that are stored in some repository. The requirements for storage systems for these Big Data applications are scalability at low cost and fast access. The faster the data is accessed, the shorter the processing time and therefore the lower the cost of execution on cloud platforms, where the pay-as-you-use model is very popular.

Data lakes are a storage solution that have emerged to optimize the performance of analytical tools such as Hadoop or Spark [60]. Data lakes are large, highly scalable storage systems that also allow data to be stored in RAW format, in such a way that it does not limit the data sources, nor does it require prior preprocessing. Data lakes are used in order to facilitate the storage of massive amounts of data and provide quick access to them when required by analytical tools. An important characteristic of data lakes is that they are built using common low-cost technologies, so that the increase in capacity does not represent an unaffordable increase in cost.

While early generations of data lakes were built as flat architectures, with a single data storage area, and Hadoop being the main tool used, they have evolved to storage systems sectioned in ponds, where data can be partitioned based on their class or the step of the analytical process where they are used. Also, more storage platforms have been included such as Amazon S3, Azure Data Lake, or similar open-source tools, like CEPH, for internal developments [124].

Regardless of the internal architecture, or the tools used for its development, fast access to stored data is one of the main requirements of data lakes. Caching and prefetching are widely used techniques to bring data closer and can be used by analytics frameworks to reduce response times to data [10]. However, access to the data stored in the data lake heavily relies on the tools used for its construction, and the data access mechanisms in the different nodes. Some of the storage tools used to build data lakes, implement caching mechanisms to improve access to data and use load balancing algorithms in order to evenly distribute requirements, and therefore the cached data, among the different nodes of the cache layer, so that response times from each of them are similar. In this chapter we propose the use of algorithms that prioritize maximizing the use of the cache (hit rates), without neglecting load balancing, as a method to obtain better access times to the data stored in the data lake.

The rest of the chapter is organized as follows, in section 7.2 we describe the architecture for a CEPH based data lake with a cache layer, in section 7.3 we show our proposed request routing algorithm, then ~~in Section~~ in section 7.4 we present the test environment used to evaluate the proposed algorithm. We analyse the evaluation results in section 7.5, and conclude in section 7.6.

7.2 Architecture

To evaluate the performance of our algorithm for request routing for a distributed cache, we propose as a test platform a data lake based on a CEPH cluster. CEPH is a popular open source distributed

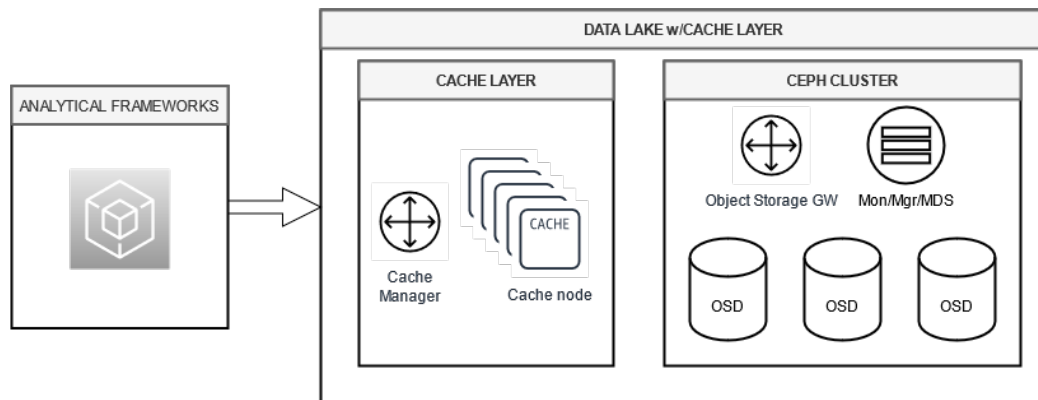

Figure 7.1: Proposed architecture for a CEPH-based data lake, acting as object storage, with a caching layer based on Nginx as cache nodes and Caddy as the cache manager or balancer; if the object storage gateway is replicated for scalability purposes, the cache nodes can be integrated or merged with the object storage gateway nodes.

storage platform [159]. One of CEPH's configurations is as an object storage system [1], which enables the storage of raw data, regardless of the information format. Additionally, CEPH provides tools that allow providing access simulating the Amazon S3 service [70], which works on the HTTP protocol and is widely used by developers.

To provide a cache layer in access to the object store, we use a farm of Nginx servers, which communicate natively (HTTP) with the S3 gateway provided by CEPH.

Finally, as manager of the cache layer we will use a Caddy server, which acts as a reverse proxy distributing the clients' requirements among the different nodes of the cache. The Caddy project implements several access algorithms, including one that seeks to optimize the load balancing between nodes and another that assigns each new requirement to the node that has the lowest load assigned (waiting requirements). Being an open-source project, the Caddy server allows us to implement not only our proposed algorithm, but also other recent algorithms such as consistent hashing with bounded loads [104].

7.3 Proposed algorithm

To improve access to data stored in a data lake containing a cache layer, we propose an algorithm that is a variant of previous work that was applied to improve task allocation on function-as-service platforms. In this chapter, we adapt our algorithm to fit the operation of a cache layer in front of an object storage using the HTTP protocol. With our algorithm, we seek to maximize the use of the cache layer, without neglecting the balanced distribution of load between the different nodes, avoiding the appearance of hot spots.

[1] https://ceph.io/ceph-storage/object-storage/

Algorithm 7: Cache-aware request routing algorithm for CEPH-based datalakes

Global data: Consistent hashing ring for cache nodes hosts, $C = c_1, ..., c_n$, Hash functions H_1 and H_2, maximum load threshold, t

Input: Requested URI including the object ID, uri

Output: Cache node A, to which uri request should be assigned

```
   /* Calculate two possible cache node targets                    */
1  t1 = H_1(uri)%|C| + 1
2  t2 = H_2(uri)%|C| + 1
   /* Select target with least load                                */
3  if (load(c_t1) < load(c_t2))then
4  |  A := t1
5  else
6  |  A := t2
   /* If target is not overloaded, we are done                     */
7  if (load(c_A) < t)then
8  |  return c_A
9  else
   |  /* Balance load                                              */
10 |  return min(length(c_i)∀i)
```

Our algorithm uses hash functions applied to the identifier of the requested object, in order to consistently redirect every request for the object to the same node of the cache layer. This way, it raises the chance for accessing an object that is already in the cache, leading to lower access latency times.

It is well known that using hash functions to dispatch object requests leads to hot spots when the popularity of stored objects is highly skewed. To mitigate this problem, we implemented two actions. First, we use the power-of-2-choices technique [105] to select, using hashing, 2 candidate nodes to receive the request, and we select the node with the lowest load, meaning the size of queued requests. Second, we use a threshold to prevent the selected node from becoming a hot spot. If the selected node has a load that exceeds the defined threshold, the request is routed to the node with the lowest load of all available nodes. This second control will direct the request to a node where the object is probably not cached yet, however, since it is the node with the least load, the response latency is expected to be low. Algorithm 7 shows our algorithm for the cache layer from a CEPH-based data lake.

7.4 Experimental design

For the experimental evaluation we built a test environment following the design described in section 7.2. The test environment was built in CloudLab [56], using 2 types of nodes: d710 nodes were used to build the object storage and the cache layer, and PC3000 nodes were used to mimic accesses from

analytical frameworks.

For the object storage layer, we deployed a CEPH cluster, with 3 OSD nodes for replicated object storage, a single node for cluster management and monitoring, and a single node acting as a gateway to access the stored objects (using CEPH's S3 API [70]). The cache layer was built using 10 Nginx servers as distributed cache nodes, and a Caddy server, acting as a reverse proxy to distribute the object requests to the cache nodes.

We implemented our algorithm as a packet routing policy in version 1.0.1 of Caddy [2], an extensible platform to run Go applications, which is commonly used as HTTP server. Our implementation has 54 lines of code. To increase the validity of our evaluation, we also implemented a routing policy based on the Consistent Hashing with Bounded Loads algorithm, using an open implementation in Go [3].

To generate the requirements or accesses to the objects in the data lake, we use KV-replay [36], which is based on YCSB [48], a popular benchmarking tool that, among its connectivity modules, includes a plugin to support the REST protocol. This setup allows us to generate queries with URIs simulating access to the Amazon S3 service.

For the experimental evaluation we use a dataset with 100K different objects, with a total size of 60GB. The objects have sizes ranging from 17 bytes to 9 MB. Nearly 20% of the objects are smaller than 100KB and 84% are smaller than 1MB. Object names were created using seeded SHA512 hashes, to avoid named based locality. This simulates an environment in which many small files are stored and accessed; a concrete example of such an environment is the case of images that need to be stored, retrieved, and analyzed.

For each experiment we generated 300K object requests. We generated requests using two approaches. First, we used a Big Data storage (HDFS) workload from Yahoo [5] to select four sections, each one with 75,000 requests records accessing 89,616 unique objects. Each section or trace was replayed by a KV-replay node with 8 threads of execution. Additionally, a synthetic workload was generated with KV-replay, using a Zipfian distribution with coefficient=0.5 to get a dataset similar to the trace from Yahoo. With this scenario we produced access to 88698 unique objects. This workload was also dispatched using four KV-replay nodes, with 8 threads each.

7.5 Results

To test the validity of the use of cache affinity policies in algorithms for load distribution in data lake storage systems, we propose the following research questions:

- RQ1: How much does the latency of the accesses improve in the data lake, when we use our proposal based on cache affinity, versus a load balancing algorithm?

[2] https://github.com/caddyserver/caddy
[3] https://github.com/lafikl/consistent

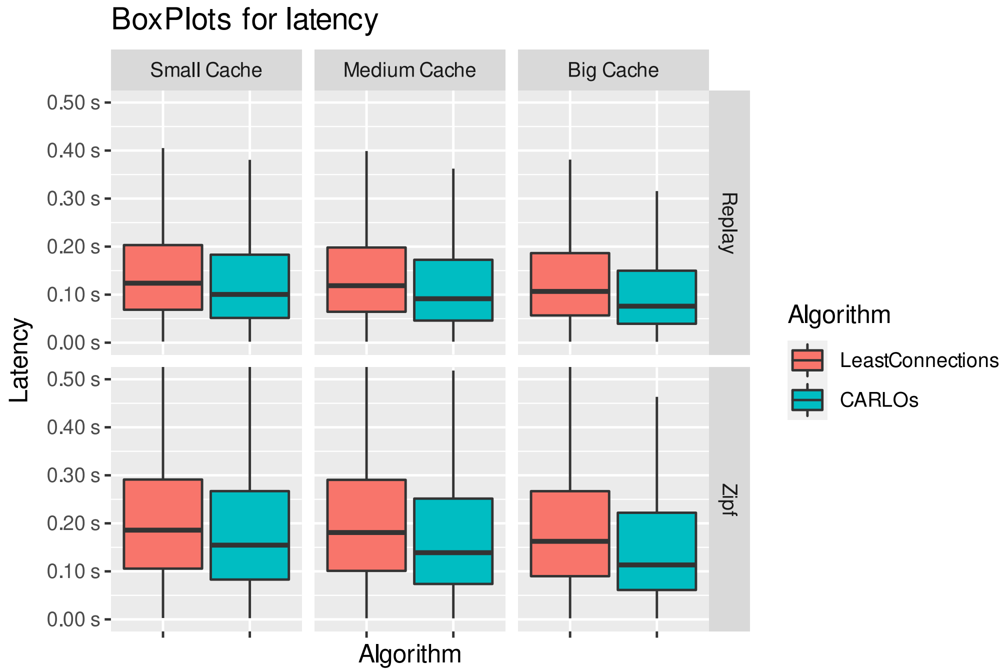

Figure 7.2: Latency Boxplot CARLOs vs LeastConnections.

- RQ2: How much does the access throughput improve in the data lake, when we use our proposal based on cache affinity, versus a load balancing algorithm?
- RQ3: Considering that there are other algorithms for request routing based on cache affinity, how does our proposal behave in comparison with other algorithms?

To answer RQ1 and RQ2, we compare the results of the experiments described in section 7.4 for the LeastConnections algorithm, which is the Caddy policy that seeks to keep the load balanced between the nodes, and CARLOs, which is our proposal to exploit the affinity of cache without causing too much imbalance in the load of the nodes.

As shown in Figure 7.2, the load balancing algorithm (LeastConnections) produces higher latencies than those obtained using our CARLOs proposal. If we compare our algorithm with a threshold of 8, CARLOs yields median latencies that are up to 20% lower, thus leading to faster access to objects. This trend continues up to the 75th percentile of the latencies, however tail latencies are better for the LeastConnections policy, where the 99th percentile on the latencies are up to 6% faster.

Figure 7.3 shows the evolution of average latency over 10-seconds windows. All cases show better performance (lower latency) for our algorithm compared with the default policy of Caddy (load balancing). We can also see the difference performance caused by the type of workload,

Table 7.1: Throughput CARLOs vs LeastConnections

Workload	**Cache Size**	**LeastConns**	**CARLOs**	**SpeedUp**
Replay	*Small*	46.34	48.76	1.05
Replay	*Medium*	46.99	50.33	1.07
Replay	*Large*	47.14	50.03	1.06
Zipf	*Small*	33.02	35.02	1.06
Zipf	*Medium*	33.16	36.23	1.09
Zipf	*Large*	34.02	35.96	1.06

despite both workloads have the same requests and a similar number of referenced unique objects. The synthetic workload (Zipfian) doesn't has temporal locality, leading to higher, but more stable, latencies. On the other hand, the replayed real trace performs better with lower latencies, as it's taking more advantage of the cache layer. Lower latencies lead to the workload reproductions in a shorter period.

For throughput evaluation, Table 7.1, and Figure 7.4 show that CARLOs algorithm obtains an improvement between 5% and 9%, depending on the size of the cache. The smaller improvement in this measure, compared to those observed in latencies, can be explained by the best performance of the LeastConnections policy in the highest percentiles of latencies.

To answer RQ3, we performed the same set of evaluations, but this time we compared the results

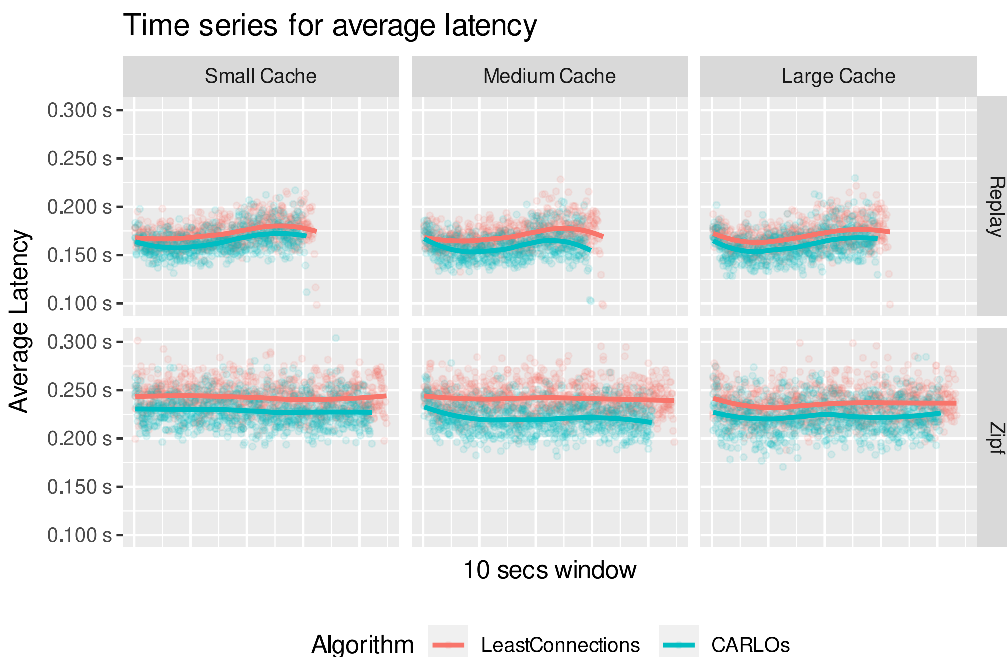

Figure 7.3: Average latency for 10-seconds windows, comparing CARLOs vs LeastConnections.

Table 7.2: Throughput CARLOs vs Other cache-affinity algorithms.

Workload	**Cache Size**	**UriHash**	**ConsistentHash**	**CARLOs**
Replay	*Small*	47.98	47.99	48.76
Replay	*Medium*	49.74	49.27	50.33
Replay	*Large*	48.58	50.06	50.03
Zipf	*Small*	34.98	34.30	35.02
Zipf	*Medium*	36.29	35.67	36.23
Zipf	*Large*	35.39	35.89	35.96

of our proposed algorithm, with two other cache affinity algorithms UriHash (which already comes as a Caddy policy) and Consistent Hashing with bounded loads (which was added as a Caddy policy as explained in section 7.4).

Regarding throughput, Table 7.2 shows that CARLOs always gets higher (better) throughput than the ConsistentHashing algorithm, but is sometimes less efficient than the UriHash algorithm. It should be noted that the results are very similar, with typical differences between 1% and 2%, so these results should not be considered conclusive.

Figure 7.5 shows the distribution of latencies obtained by our CARLOs algorithm, compared

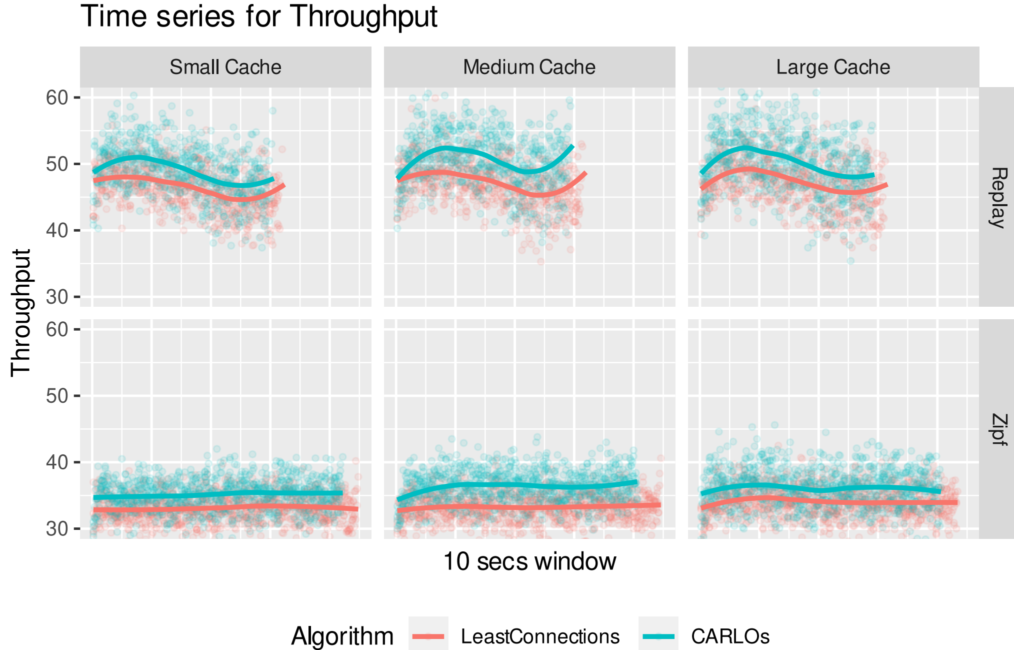

Figure 7.4: Throughtput for 10-seconds windows, comparing CARLOs vs LeastConnections. This figure shows that the performance of CARLOs algorithm is consistently better alongside time and different cache sizes.

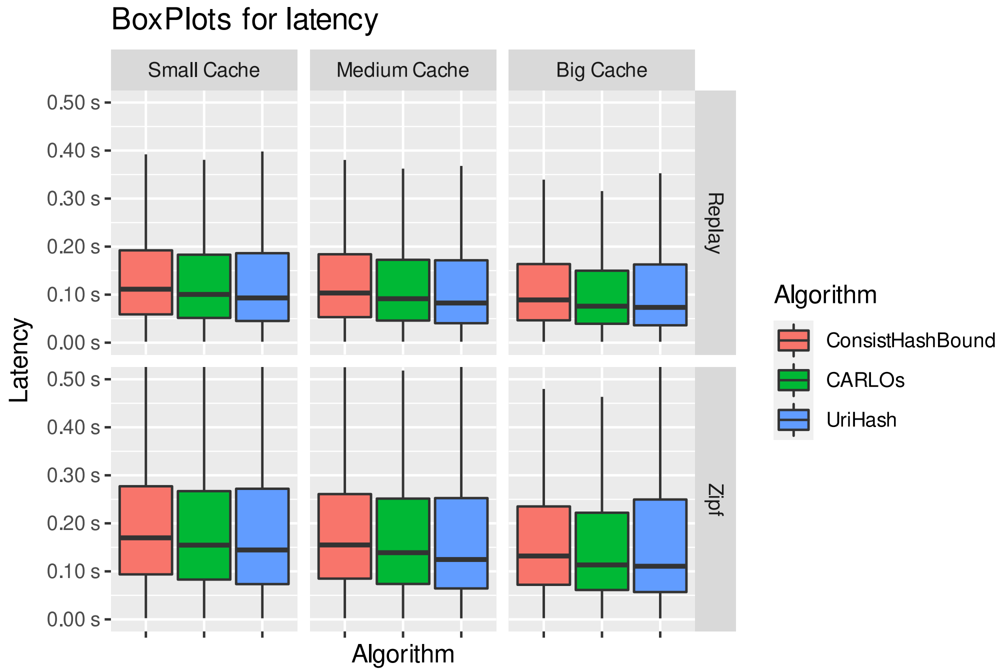

Figure 7.5: Latency Boxplot CARLOs vs Other Cache affinity algorithms.

with other cache affinity based policies. The UriHash policy, which is included in Caddy, produces a better median latency, however, the bodies of boxplots show a higher dispersion of the results. This is confirmed with the IQR (interquartile range) value, which is up to 17% shorter for CARLOs. Comparing CARLOs with the policy based on the Consistent Hashing with Bounded Loads algorithm, we find that the dispersion is similar, having practically similar interquartile ranges (IQR). However, the latencies are lower for CARLOs for the percentiles analyzed.

Statistically meaningful evaluation

For ensure a statistically meaningful evaluation, we used an one-way ANOVA test, and complemented this evaluation with the Tukey HSD (Tukey Honest Significant Differences) test for performing multiple pairwise-comparison of CARLOs with other cache-affinity based policies. Before running the ANOVA evaluation, a normality test was run on all the measurement sets using the Lilliefors (Kolmogorov-Smirnov) normality test.

The ANOVA test was performed on the *throughput* and *average latency* values for all the experimental scenarios, and results were consistent along all of them. We present results for the *Large Cache* scenario, for both *Replay*, and *Zipfian* workloads. Table 7.3 summarizes the p-values for the Anova test. We can see that, for every scenario, and both for Latency and Throughput, there is a

Table 7.3: ANOVA Test Results for statistically meaningful evaluation between CARLOs and other request routing policies.

Compared pair	**Workload**	**Metric**	**p-value**	**Statistically different?**
CARLOs vs LeastConnections	*Replay*	Latency	$7.86e-14$	True
CARLOs vs LeastConnections	*Replay*	Throughput	$3.39e-11$	True
CARLOs vs LeastConnections	*Zipf*	Latency	$< 2e-16$	True
CARLOs vs LeastConnections	*Zipf*	Throughput	$< 2e-16$	True
CARLOs vs UriHash	*Replay*	Latency	0.0001	True
CARLOs vs UriHash	*Replay*	Throughput	0.00009	True
CARLOs vs UriHash	*Zipf*	Latency	0.0005	True
CARLOs vs UriHash	*Zipf*	Throughput	0.0002	True
CARLOs vs ConsistHashBound	*Replay*	Latency	0.9	False
CARLOs vs ConsistHashBound	*Replay*	Throughput	0.98	False
CARLOs vs ConsistHashBound	*Zipf*	Latency	0.91	False
CARLOs vs ConsistHashBound	*Zipf*	Throughput	0.75	False

statistically significant difference between CARLOs and LeastConnections, and the same between CARLOs and UriHash, the two well-known policies evaluated. However, there is not a statistically significant difference between CARLOs, and Consistent Hashing with Bounded Loads, another state of the art policy based on cache affinity rules.

7.6 Conclusions

Exploiting cache affinity to improve requests latency in data lake storage systems is plausible. Our work shows that using cache affinity policies in the caching layer for data lake systems is better than using a more common load balancing policy, where new requests are sent to the host with lowest active requests.

Our algorithm outperforms the standard LeastConnections algorithm, reducing request latency by up to 20%, and increasing the throughput between 5% and 9%.

The request proxy used in our experiments, which is the node in charge for request routing decisions, already includes a cache affinity based policy (UriHash), however, our proposed algorithm also reduces the latency and increases the throughput.

We also compared our solution with another state-of-the-art algorithm for load balancing, Consistent Hashing with Bounded Loads, but our experimental results doesn't show statistically significant differences for average latency, nor for average throughput. However, experimental results show that CARLOs, our proposed algorithm, got shorter IQR for request latencies, which can be interpreted as a greater degree of stability in the response times obtained.

Chapter 8

Serverless-based microservices to support self-adaptive cloud services

The research community has made significant advances towards realizing self-tuning cloud caches; notwithstanding, existing products still require manual expert tuning to maximize performance. Cloud (software) caches are built to swiftly serve requests; thus, avoiding costly functionality additions not directly related to the request-serving control path is critical. We show that serverless computing cloud services can be leveraged to solve the complex optimization problems that arise during self-tuning loops and can be used to optimize cloud caches, *for free*. To illustrate that our approach is feasible and useful, we implement SPREDS (Self-Partitioning REDiS), a modified version of Redis that optimizes memory management in the multi-instance Redis scenario. A cost analysis shows that the serverless computing approach can lead to significant cost savings: The costs of running the controller as a serverless microservice is 0.85% of the cost of the always-on alternative. Through this case study, we make a strong case for implementing the controller of autonomic systems using a serverless computing approach.

The work presented in this chapter was first published at MDPI Computers journal [35].

8.1 Introduction

Application-controlled cloud caches implemented with fast in-memory key-value stores, like Redis and Memcached, have become ubiquitous in modern web architectures [1]. Content providers use caches to reduce latency and increase throughput, increase user engagement and profits, and reduce infrastructure costs [1, 134]. Proper configuration and tuning of these caches is critical, as sub-optimal configurations lead to increased miss rates and a resulting penalty in end-to-end performance, negatively affecting the business goals: It has been reported by Amazon that a 100 ms latency penalty

can lead to a 1% sales loss, and by Google that an additional 400 ms delay in search responses can reduce search volume by 0.74% [134].

A time-tested way to improve cache performance is to optimally partition the cache [15, 142, 99, 139, 122, 58, 137, 45, 82, 46, 29, 47, 1, 83]; for example, by dynamically partitioning the total memory between users or applications. However, current cloud caches support only static partitioning while others provide no control over the partitioning. Redis[1] is an example of the former, while Memcached is of the latter. While several solutions targeting cloud caches have been proposed [137, 45, 82, 46, 29, 47], these have not been added to industry-grade software caches due to performance concerns.

In this chapter, we present a serverless computing architecture using the Function-as-a-Service model, to optimize cloud caches *for free*. We implement this solution in SPREDS, a Self-Partitioning REDiS. SPREDS leverages modern data structures and statistical sampling methods to efficiently obtain online estimates of the real miss rate curves (MRCs) [107]. The backend performance profiles and MRCs are combined into a utility function to maximize. The resulting optimization problem is solved outside the cache using a serverless computing approach. Our experimental results show that the performance overhead due to monitoring the cache is low and that implementing the autonomic controller as a serverless microservice is feasible and useful. We present cloud cost calculations that show that the invocations to the controller are either free or very cheap, with the costs of running the controller as a serverless microservice being just 0.85% of the cost of the always-on alternative. Additionally, the optimization problem is solved outside of the machine running the cache and does not consume resources of the caching nodes.

Using SPREDS as a case study, we argue that the proposed approach should be considered in future autonomic and self-* systems, as it is a low-cost and low-overhead way to calculate complex adaptation decisions applicable to systems running on public cloud providers. We end by outlining some challenges in implementing this vision, and discuss how to address them.

8.1.1 Contributions and chapter roadmap

This work makes the following contributions:

1. We survey the most current research in the domain of self-tuning cloud caches, and study the designs used by prior approaches as a way to motivate the need for a more modern, cheaper, cloud-native solution (Section 8.2).

2. We re-visit the *memory partitioning problem* and model it as a mathematical optimization problem with restrictions that are specific to the multi-instance cache on a shared node, studied in this chapter (Section 8.3).

3. We present a novel design for implementing autonomic cloud caches that leverages serverless

[1] Redis stands for Remote Dictionary Server.

computing cloud offerings to implement the autonomic controller (optimization module) at a low cost (Section 8.4).

4. We show the feasibility of our approach through implemeting SPREDS, a real implementation of our design using Redis and AWS Lambda (Section 8.4).

5. We present real experimental results and corresponding cost analysis, that validate the usefulness of our proposal (Section 8.5).

6. We make a case for adopting our serverless approach to other autonomic, self-tuning systems, and study the challenges in realizing this vision (Section 8.6).

8.1.2 Threats to validity

We assume that the mathematical optimization problem to find the optimal memory partitioning can be solved in a reasonable amount of time (e.g., takes no more than 25% of the time between adaptation cycles). If solving the optimization problem takes too long, the solution proposed in this chapter is not applicable.

Our solution relies on estimated miss rate curves, as getting real MRCs with a low performance overhead is—to the best of the state-of-the-art knowledge in caching [156]—not possible. However, the results we present from real cloud experiments show that the accuracy of the MRCs is good enough for our purposes: SPREDS can improve performance over a static partitioning approach. Nevertheless, it is possible that there may exist workloads for which the estimated curves are not good proxies of the real ones.

Automatic parameter tuning depends on being able to monitor dynamic behavior at a granularity that is useful for making informed predictions of the impact of changing specific parameters. Our solution leverages recent advances in efficient MRC estimation [107] that apply only to the caching domain. An alternative approach to be considered for future work is constructing a utility function that relies only on the metrics and information that can be obtained from system logs; this would be a zero-overhead approach, as many useful information from each system component is typically already being logged, stored, and monitored—following a modern DevOps mindset [57, 23].

Finally, we argue for using a serverless computing approach in our design and present cost estimates that are considerably cheaper than the alternative of an always-on service. Whether this is actually true or not in practice, depends on: (1) how frequently is the optimization service invoked, and (2) the costs of each of the cloud services used in its implementation. We believe the serverless approach may be the cheapest option for small and medium-sized organizations in the near future but cheaper alternatives may arise in the long run or may already be available for larger organizations for which an always-on service is likely a better alternative.

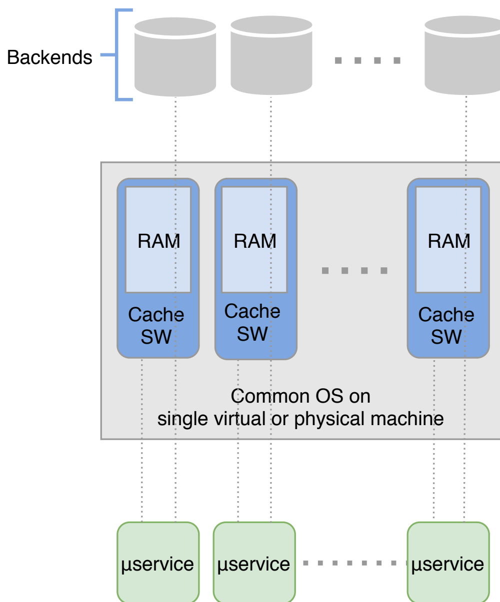

Figure 8.1: Illustration of the motivating architecture used in this chapter. The figure shows a web or mobile application with a microservices architecture. There is a caching layer that speeds up accesses to data stored in several heterogeneous backends.

8.2 Background, motivation and related work

Key-value stores are primitive databases that support efficient insertion and lookup of data indexed by user-defined keys. In-memory key-value stores work like remote hash tables or dictionaries, and can answer requests with very high throughput and low latency. At the time of this writing, the most common in-memory key-value store software products are Redis[2] and Memcached[3] [51]. These products are frequently used as caches at the backend of modern web and mobile applications. Caches implemented with in-memory key-value stores are not transparent caches; the cache is invoked explicitly in the applications, serving complex business logic workflows. In this chapter we study Redis because it is the most popular key-value store as-of December 2019 [51].

We show a common use case of cloud caches in Figure 8.1: A system that stores information—like product inventory, user profiles, and session information—in different storage backends. There is a frontend that aggregates information from these backends and presents it to the user. To improve latency and reduce pressure on the storage backends when serving user requests, the frontend typically first contacts a caching layer seeking the required information. If the data being sought is not stored in the cache—a *cache miss*—the application needs to contact the specific storage backend, get the data and return it to the user. The application then contacts the cache to store a copy of the

[2]https://redis.io
[3]https://memcached.org

data, so that future requests can be served directly from the cache. Each of these storage backends have their own unique performance profiles. For example, the databases supporting complex queries are much slower than object storage devices that store image files. The following is a small list that exemplifies the type of things being cached in the backends of web or mobile applications:

- User profiles, where profiles contain data pulled from several other backend systems.
- Tracking information, like user engagement counters.
- User avatars or other images shown in the frontend to the end users.
- Business reports resulting from querying multiple tables in an OLTP database.
- Session information, like the per-user application system state.
- User status updates in social networks; users read the updates of their "friends".

One important factor in the performance of a cache is how much memory it has available to store objects: The larger the cache, the higher the percentage of objects from the storage backend that it can hold, and thus, the higher the likelihood of a request being a hit and not a miss.

The cache's eviction policy determines which objects are removed from the cache to make space for new objects being added. Cloud caches support several eviction algorithms, with least recently used (LRU) or some variant of this algorithm being the most commonly used one, as it is known to perform well for a wide variety of workloads. LRU assumes that recent past behavior is a good predictor of future behavior. When an object is to be evicted, it selects the object that has been used least recently because this is the one that it predicts is the least likely one to be re-used in the near future. By default, Redis uses a randomized version of LRU [120] which samples x objects and evicts the one that was accessed least recently; x is configurable and set to five by default.[4]

The other important factor influencing the performance of a cache is the workload characteristics [8, 36] like the skewness of the distribution of the popularity of the objects, and the degree of temporal locality present in the accesses to the objects in the cache. These workload characteristics are application-dependent and may be dynamic; thus, the caches must be able to self-adapt to these changes.

8.2.1 How important is (optimal) memory partitioning in cloud caches?

To make best use of CPU resources, modern microservices architectures favor shared-nothing and stateless approaches to distributed systems. A common scenario is for multiple Redis instances to be co-located in the same machine (see Figure 8.1). Each of these instances serves a different workload or application, and can be configured independently. In this scenario, the machine's memory becomes an important resource that must be adequately partitioned between the cache instances.

Table 8.1: Summary of the recent work in the domain of self-partitioning cloud caches. For each of the papers, we highlight one result demonstrating the achievable performance gains. Prior solutions differ due to differences in methods, service-level objectives, workloads and experimental configurations.

System	**Open Source?**	**Goal**	**Result**
Moirai [137]	Prototype	Supports diverse SLOs	Overall throughput increases by more than 2.5x, in a multi-tenant datacenter.
Dynacache [45]	No	Minimize miss ratio	Reduces number of misses by more than 65%.
LAMA [82]	No	Minimize miss ratio	Reduces average miss ratio by 41.9%, saving 40.8% memory space.
Cliffhanger [46]	No	Minimize miss ratio	Reduces number of cache misses by 36.7%.
RobinHood [29]	Testbed	Minimize request P99	Meets 150ms P99 goal 99.7% of time (vs 70% of time for next best policy).
Memshare [47]	Simulator	Minimize miss ratio	Increases combined hit rate from 84.7 to 90.8%.

We surveyed recent papers tackling the problem of self-partitioning cloud caches [137, 45, 82, 46, 29, 47] and found that intelligent cloud cache partitioning can lead to significant performance gains: Depending on the service-level objectives and resulting utility function, these improvements can be in terms of higher throughput, higher cache hit rate, and reduced end-to-end latency. Tables 8.1 and 8.2 summarize our observations. Based on the results presented by the prior studies and on our own observations, we posit that *smart and adaptive memory partitioning in caches supporting different workloads leads to significant performance gains.* Sadly, memory partitioning in Redis is currently only static and manual. The case of Memcached is even worse, as it gives more memory to the application issuing more requests, regardless of whether this is good for the overall system [121].

8.2.2 Architectures for self-partitioning cloud caches

Self-adjusting capabilities that involve solving complex optimization problems can impose unnacceptable performance overheads on the system being tuned. We argue that a serverless architecture approach can be used to overcome this limitation. To better explain why this is the case, we first study the architectures used in prior self-partitioning caches. We summarize their architectural decisions regarding where to locate the monitoring engine and the autonomic controller in Table 8.2 and analyze the limitations of these architectures next.

Prior projects have used one of the following approaches to limit the performance overhead

[4]Since version 3, the LRU implementation of Redis also takes a pool of good candidates for eviction.

Table 8.2: Architecture of the self-partitioning caches proposed in recent literature. MR: Miss rate.

System	Location of metrics engine (monitoring agent, MA), and Location of optimizer (autonomic controller, AC)
Moirai	MA: Local hypervisor module analyzes workloads on system. AC: Centralized controller, external to cache; shared at datacenter level.
Dynacache	MA: Offline external; heavyweight stack distance algorithm. AC: In-cache LP solver that assumes convexity of MRCs.
LAMA	MA: In cache metrics (independent thread). AC: In-cache algorithm (dynamic programming algorithm).
Cliffhanger	MA: In cache; approximates MR gradients w/ shadow queues. AC: In-cache; incremental *cliffhanger* algorithm.
Memshare	MA: In cache *arbiter* approximates MR gradients. AC: In-cache arbiter uses MR gradients to incrementally adjust partition sizes.
RobinHood	MA: External RBC server; shared between applications. AC: Distributed controller; one per caching server.
SPREDS	MA: Local to cache; external process. AC: Pay-per-use service on a serverless computing platform.

introduced by solving the optimization problem:

A1: Make small, gradual, changes to the memory allocations exploring the configuration space to find the optimal partitioning. This approach removes the CPU cost of solving the optimization problem but introduces frequent resizing costs as the internal state and data structures of the cache need to be modified for each small step in the exploration of the configuration space. Cliffhanger [46] is an example of a system that makes small, continuous, costly changes.

A2: Simplify the problem by limiting the number of partitions and making other (unrealistic) assumptions about the workload, thus ensuring that solving the optimization problem becomes tractable without incurring in high computation costs. This approach has been suggested as a way to solve the optimization problem within the cloud cache but without consuming excessive CPU resources which would slow down the cache requests. Dynacache [45] is an example that simplifies the problem to make it more tractable.

A3: Take the solving of the optimization problem outside of the caching software and move it to an external controller. This is the most common approach in the literature of self-adaptive systems (e.g., see [147, 1, 91]).

Approach A3 is the better option because it does not incur frequent resizing costs (limitation of A1) nor does it make unrealistic assumptions about the workload (limitation of A2). For this reason, we opt for approach A3 in SPREDS.

It should be noted that, while A3 overcomes the performance and accuracy limitations of A1 and A2, it may incur additional costs related to running the external controller. We are not aware

of prior work comparing the monetary costs of running an external controller. These costs can differ depending on the architectural approach used to implement A3 in a real system:

(a) Always-on shared service: An always-on shared cloud adaptation service can be offered for free or at a reasonable cost by the cloud or a third-party provider. The costs are shared by tenants or absorbed by the provider seeking a competitive advantage. An example of a recent product in this domain is the self-indexing service for the Microsoft Azure SQL DB [50]. However, this approach lacks flexibility and does not encourage innovation in the tuning algorithms (the service provider controls the algorithm and tenants cannot test improved adaptation algorithms). Furthermore, some utility functions—like those that seek to minimize operational costs—may go against the provider's business interests and the providers would not offer them to their tenants.

(b) Always-on client-managed service: The cloud tenant runs the controller as one more always-on service in their system. For example, in the database domain, OtterTune [147] has been proposed as an external database tuning system that can tune the performance of databases as well as external human experts. The costs of the always-on client-managed service approach can be too high for small or medium organizations, if the service is infrequently used. In addition, it has the added overhead of having to manage an additional online service, and thus, constitutes an option only suitable for large enterprises.

(c) Serverless microservice: The approach argued for in this chapter is deploying the controller as a serverless microservice, using a Function-as-a-Service (FaaS) offering like AWS Lambda[5] or Azure Functions[6]. FaaS offerings are gaining increasing attention in the community as they let tenants run code without provisioning or managing servers, and paying only for the compute time they consume [148]. The solver is an external process owned by the client and implemented using a serverless architecture. This on-demand approach avoids service over-provisioning and reduces the costs of operating the controller. Furthermore, it lets the tenants tailor the utility function and optimization-solving algorithm according to their specific needs.

The details of our proposal are presented in Section 8.4. In Section 8.5 we experimentally validate the approach and present a cost analysis that shows that the always-on client-managed service approach is much more expensive than the serverless approach advocated for in this chapter.

8.2.3 Why aren't current cloud caches already self-partitioning?

Some of the projects described in Table 8.1 involved collaborations with industry—namely, Facebook, Akamai, and Microsoft. It is possible that these (and other) companies have self-partitioning cloud caches being used in production. However, these adaptation mechanisms have not been added to the open source versions of Redis or Memcached.

[5]`https://aws.amazon.com/lambda/`
[6]`https://azure.microsoft.com/en-us/services/functions/`

Approaches that gradually explore the configuration space hoping to find an optimal solution (e.g., [46]) incur in a high penalty when the cost of reconfiguration is not negligible, as has been observed in the current implementation of the resizing mechanism of Redis [1]. Furthermore, these solutions are tailored for a specific solution and do not scale to self-tuning other knobs of the caching software. A better approach is utility-driven approaches which are flexible and can support diverse service-level objectives. However, these are deemed unscalable due to the high CPU consumption of the algorithms used to solve the optimization problem [71]. We believe the serverless architecture proposed in this chapter can overcome both challenges and can facilitate adding self-adaptation functionality to cloud caches and other software in the near future.

8.2.4 Other related work

Earlier in this Section we analyzed the most relevant prior work in self-partitioning cloud caches; the results of our analysis was presented in Tables 8.1 and 8.2. In this subsection we discuss other prior research that is related to SPREDS.

Other uses of workload-driven partitioning schemes for caches include partitioning flash-based caches into hot/cold areas to support efficient data compression [85] or to determine the right number of replicas of each partition [167]. In addition to improving performance, others have included the notions of fairness, isolation and strategy proofness in their partitioning schemes [121, 165]; SPREDS was designed for the case in which the applications belong to the same tenant and do not try to game the system. Another consideration that can be incorporated into the optimization problem is the penalty associated with memory re-allocation during partitioning cycles; solutions like the one used in pRedis [117], that factor this into the optimization problem, are orthogonal to the architectural approach proposed in this chapter.

A few self-tuning database products and services have been proposed by academia and industry. OtterTune [147] is a tuning service for MySQL and PostgreSQL that automates the process of finding good settings for a database's configuration, reusing training data gathered from previous tuning sessions. Das et al. [50] describe an auto-indexing service for Microsoft Azure SQL Database. Oracle recently released their Autonomous (cloud) Database[7] and ScyllaDB offers a database with limited self-managing properties[8]; however, the latter is a rule-based tuning solution [14].

Idreos et al. [84] recently introduced the concept of design continuums for the data layout of key-value stores and present a vision of self-designing key-value stores that automatically choose the right data layout for a specific workload and memory budget. This is a related but different problem than the one used as a case study in this chapter; both self-* approaches can co-exist in the same key-value store.

In general, the idea of offering adaptations "as a service", as a path to making autonomic systems

[7] `https://www.oracle.com/database/autonomous-database.html`
[8] `https://www.scylladb.com/`

a reality, has been argued before [1, 91]. However, a traditional always-on service makes sense when the service is provided by the cloud or third party provider, or for large organizations. In this chapter, we propose a variant of this idea, with a serverless microservice architecture and provide a proof-of-concept implementation highlighting its usefulness.

8.3 The memory partitioning problem

We consider a system where a virtual or physical machine hosts n instances of the caching software, each serving a different application as depicted in Figure 8.1; these applications compete for the allocation of the total memory, M. The amount of memory assigned to each application i is denoted by m_i. Our model also works for a multi-tenant architecture, as long as the different applications sharing the cache belong to the same organization. Table 8.3 contains a reference of the parameters used in the model definition.

Table 8.3: Parameters used in the model definition.

Parameter	**Description**
n	Number of instances of the cache running on the system.
i	Identifies an specific application or workload.
M	Total system memory.
m_i	Memory assigned to application i.
$\underline{m}_i$	Minimum memory assignment for application i.
bd_i	Access latency of backend system, including the time to process a cache miss.
cd_i	Access latency of the caching system.
U_i	Individual utility function of application i.
w_i	User-defined weight for application i.
f_i	Access frequency of application i.
EAT_i	Effective access time of application i.

When application i needs some data, it first looks for it in the cache. If the sought data is not there—i.e., a *cache miss* occurs—the application obtains the desired information from its corresponding storage backend. Each backend has its own performance profile; i.e., its own average latency to access the backend bd_i. For example, a database server used to build user profiles may likely be slower answering requests than a service that generates unique user identity avatars based on the client's username or IP address (e.g., like Github's identicon).

After a cache miss, the data is typically inserted in the cache (after making space for it by evicting less valuable data, if necessary) so that it will be available at the cache for future requests. However, as these are explicit—not transparent—caches, the application is free to implement more complex admission logic.

We consider memory as the only shared resource, ignoring the sharing of CPU. In-memory key-value stores are memory- and not CPU-bound. This has been reported by Redis[9], and observed in real Memcached deployments [46].

In this work, the goal is *Pareto efficiency*: Fully utilize the memory, compute the ideal memory allocation $\mathbf{m} = [m_1, \ldots, m_n]$ and achieve the highest overall utility, given individual utility functions U_i's and total memory constraint M. This can be expressed as the following optimization problem:

$$\begin{aligned} \underset{\mathbf{m}}{\text{maximize}} \quad & \mathcal{F}(\mathbf{m}) = \sum_{i=1}^{n} w_i U_i(m_i) \\ \text{subject to} \quad & \sum_{i=1}^{n} m_i \leq M, \\ & m_i \geq \underline{m}_i, i = 1, ..., n, \end{aligned} \tag{8.1}$$

where $U_i(m_i)$ is the utility function of application i as a function of its assigned memory m_i, and configured w_i (which lets us indicate that one application is more or less important). $\underline{m}_i$ is the minimum memory assignment for application i; it can be set to zero if it is OK for the system to decide not to cache the objects of some application.

We assume a non-adversarial model in which the applications are not trying to game the system. This is a reasonable assumption when all the applications belong to the same cloud client. Given that we consider a non-adversarial model, we do not seek *strategy proofness* [121]. Some application could be able to issue workloads that lead to a higher memory assignment to said application, but this would be at the cost of reduced overall system performance.

For the utility function, we consider improving average access latency to objects in the cache to be our most important optimization metric. For eviction algorithms that fulfill the *inclusion principle* [101]—e.g., LRU—giving more memory to the cache means that the hit rate will either stay the same or will improve. Thus the way to optimize the performance for a single application is to give it as much memory as possible. As the memory is a shared resource that the applications are competing for, we devise a utility function that combines each of the utilities as a function of how much memory we have assigned to the application $U_i(m_i)$, weighed by a user-defined weight (w_i) so that the tenant can declare that improving the performance of one application is more important than the others.

We posit that improving the hit rate of one application may not be as useful as improving the hit rate of another application, due to differences in the performance profiles of the corresponding backends. For example, all other things being equal, if one application has a slow backend (e.g., one that processes slow, multi-table, SQL queries), increasing the hit rate of the cache serving that application is more useful than increasing the hit rate of a cache serving a fast backend (e.g., a NoSQL database that stores session information). For this reason, for each application we calculate

[9] `http://redis.io/topics/faq`

its *effective access time (EAT)* and weigh it by the inverse of the application's access frequency (f_i). The formula for the EAT is calculated using the classic approach devised for two-level memory systems [136], where we consider the access latency to the two levels—the cache and the backend storage—and the probability that an object or data item will be found in the fast memory tier, which in our case, is given by the cache hit rate. The access latency to these two levels is denoted by cd_i (object latency in the caching system) and bd_i (object latency in the backend system, including the time to process a cache miss). For a given application, the cache hit rate is a function of how much memory the application has been assigned $h_i(m_i)$, and can be directly obtained from the application's miss rate curve (MRC). In other words, the EAT_i is the time that it takes, on average, to access an object in application i. We define the following per-application utility function:

$$\begin{aligned} U_i(m_i) &= -f_i \times EAT_i(m_i), \text{ where} \\ EAT_i(m_i) &= h_i(m_i) \times cd_i + [1 - h_i(m_i)] \times bd_i. \end{aligned} \tag{8.2}$$

where $h_i(m_i)$ is the hit rate of application i as a function of assignment m_i, and f_i is the frequency of requests of i.

8.3.1 Solving the optimization problem

Consider problem (8.1): If the U_i's are quasi-linear the resulting optimization can be solved by solving a sequence of feasibility problems, with a guaranteed precision of ϵ in $\lceil \log_2 R/\epsilon \rceil$ iterations, where R is the length of the search interval. If the U_i's are concave, we have a convex optimization problem easily solved with off-the-shelf solvers; for example, using gradient-based methods. When the U_i's are: discontinuous, non-differentiable, or non-convex, alternative approaches are required.

One alternative is to use a probabilistic search, in which a model generates candidate points in the search of an optimum. An adaptive mechanism may be added to the generative model to improve the performance of the sequentially generated candidates. We proposed one such approach to memory partitioning in earlier work [1] and implemented it in SPREDS (see Algorithm 8). This genetic algorithm works for any partitioning problem, as it makes no assumption on the shape of the utility curves. We next describe this approach, which is one of the two solvers implemented in SPREDS.

Let $\mathbf{x} = [x_1, \ldots, x_n]$ with $x_i = (m_i - \underline{m}_i)/(M - \sum_i \underline{m}_i)$, satisfying $\sum_i x_i = 1$ and $0 < x_i < 1$. We assume that the variability of $\mathbf{x}$ can be well modeled by a Dirichlet[10] distribution $Dir(\mathbf{x}|\alpha)$, with

[10] The Dirichlet distribution $Dir(\mathbf{x}|\alpha)$ has a density function:

$$f(\mathbf{x}|\alpha) = \frac{\Gamma(\sum_{i=1}^{n} \alpha_i)}{\prod_{i=1}^{n} \Gamma(\alpha_i)} \prod_{i=1}^{n} x_i^{\alpha_i - 1}, \tag{8.3}$$

expected value $E[x_i] = \alpha_i/A$ and variance $V[x_i] = \alpha_i(A - \alpha_i)/(A^2(A+1))$, where $A = \sum_i \alpha_i$.

parameter vector $\alpha = [\alpha_1, \ldots, \alpha_n]$. Then, we define:

$$\tilde{\mathcal{F}}(\mathbf{x}) = \mathcal{F}\left((M - \sum_i \underline{m}_i)\mathbf{x} + \underline{\mathbf{m}}_i\right), \tag{8.4}$$

where $\underline{\mathbf{m}}_i = [\underline{m}_1, \ldots, \underline{m}_n]$.

We propose the following general approach, inspired in evolutionary strategies, for solving problem (8.1) in the case of non-convex and non-quasi-linear functions U_i's. We begin by setting α_i to $1/n$, for all i. We then generate K points $\mathbf{x}_k^\star|\alpha \sim Dir(\mathbf{x}|\alpha)$ for $k = 1, \ldots, K$. Note that points generated in this way satisfy all restrictions of problem (8.1).

Using points in set $\{\mathbf{x}_k^\star\}_{k=1}^K$ we construct the following prior mixture density for α:

$$g(\alpha; \{\mathbf{x}_k^\star\}) = \frac{1}{Z} \sum_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \phi_{\mathbf{x}}(\alpha), \tag{8.5}$$

where Z is a normalization constant and $\phi_{\mathbf{x}}$ is a non-negative function with finite mass concentrated around $\mathbf{x}$ (e.g., a *radial basis function* centered at $\mathbf{x}$). We proceed by sampling α from g and generating points $\mathbf{x}|\alpha_\gamma \sim Dir(\mathbf{x}|\alpha_\gamma)$, where α_γ is the vector with elements $\gamma\alpha_i$ for $\gamma > 1$. Note that, while random variables $\mathbf{x}|\alpha$ and $\mathbf{x}|\alpha_\gamma$ have the same expected value, γ has the effect of reducing the variance of $\mathbf{x}|\alpha_\gamma$ by a factor of γ^{-1}.

The above generative procedure corresponds to the following Bayesian hierarchical structure:

$$\begin{aligned} \alpha &\sim G(\alpha; \{\mathbf{x}_k^\star\}) \text{ and} \\ \mathbf{x}|\alpha_\gamma &\sim Dir(\mathbf{x}|\alpha_\gamma), \end{aligned} \tag{8.6}$$

where G is the distribution function corresponding to mixture density g.

Our method proceeds by alternatively generating parameters α—the exploration stage—and generating J points $\mathbf{x}$'s conditioned on α_γ—the exploitation stage. The prior distribution for α is then updated using the K best cumulatively-observed points $\mathbf{x}_k^\star$'s and the procedure is repeated until a satisfactory solution is found. Algorithm 8 provides the details of the proposed procedure.

We also implemented a hill climbing algorithm combined with a LookAhead approach [122][11]. SPREDS considers the number of partitions and chooses the solving method depending on the complexity of the problem. Based on experimental results presented in Section 8.4.3, we heuristically choose between both algorithms as follows. For deployments with a few partitions ($i \leq 7$), we use the proposed genetic algorithm. For deployments with a larger number of partitions, we use the hill climbing solver. Intelligently choosing the solver is out of the scope of this chapter [119, 1].

[11] We added the LookAhead approach so that this algorithm can deal with non-convex utility curves; the genetic algorithm is applicable to any shape of the utility function

Algorithm 8: Probabilistic adaptive search

Input: Functions U_i's; number K of points to use; number J of rounds; function $\phi_{\mathbf{x}}$

Output: Best point $\mathbf{x}^*$, such that $\tilde{\mathcal{F}}(\mathbf{x}^*) \geq \mathbf{x}$ for every point $\mathbf{x}$ generated

1 $\alpha_i := 1/n$ for $i = 1 \ldots, n$

2 Generate $\mathbf{x}_k^\star | \alpha \sim Dir(\mathbf{x}|\alpha)$ for $k = 1, \ldots, K$

3 **repeat**

4 Generate $\alpha \sim G(\alpha; \{\mathbf{x}_k^\star\})$

5 **for** $(j = 1, \ldots, J)$**do**

6 Generate $\mathbf{x}|\alpha_{\gamma(j)} \sim Dir(\mathbf{x}|\alpha_{\gamma(j)})$

7 **if** $(\tilde{\mathcal{F}}(\mathbf{x}) > \min_k \{\tilde{\mathcal{F}}(\mathbf{x}_k^\star)\})$**then**

8 $\{\mathbf{x}_k^\star\} := \mathbf{x} \cup \{\mathbf{x}_k^\star\} \setminus \arg\min_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \tilde{\mathcal{F}}(\mathbf{x})$

9 **until** (*Satisfactory solution* $\mathbf{x}^\star = \arg\max_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \tilde{\mathcal{F}}(\mathbf{x})$ *is found*);

10 **return** $\mathbf{x}^\star$

8.4 Design and implementation of SPREDS

We map our design of a self-partitioning cloud cache to a MAPE-K loop [22], with its corresponding monitor, analyze, plan and execute functions.

The **monitor**ing component runs in the same machine as the cache, and can be part of the cache or an external sidecar [129] microservice. This component uses statistical sampling techniques to minimize the monitoring impact. The monitor stores every N observations on a configurable cloud location. When a new set of observations is stored, a *calculate new adaptation* event is triggered. The size of the set of observations, N, is configurable and lets the user decide how frequently the system adaptations should be triggered—i.e., how frequently to calculate the optimal memory configuration and to re-partition the cache memory if necessary. Criteria for choosing N include the cache throughput and how dynamic the application workloads are.

The controller is implemented as a serverless microservice. It performs the **analyze** phase when launched in response to a *calculate new adaptation* event. In the SPREDS implementation, this microservice solves the mathematical optimization to determine how to best partition the cache memory according to the specific set of observations captured by the monitor.

The controller generates an adaptation **plan** to implement the solution, and stores it on a specific cloud storage location. The action of storing the adaptation plan on the cloud storage triggers an *execute adaptation* event which is then received by the cache. When **execute**d, this plan re-partitions the cache according to the most current (optimal) solution.

In our design, the **knowledge source** are one or more locations (e.g., buckets or directories) in a cloud storage system managed by the provider. This is where the system stores the captured metrics, monitoring data and adaptation plan. Alternatively, one or more specialized databases could be used for this purpose. For example, the Prometheus[12] time series database is commonly used in cloud-native systems to store metrics analyzed by DevOps teams.

[12]`https://prometheus.io/`

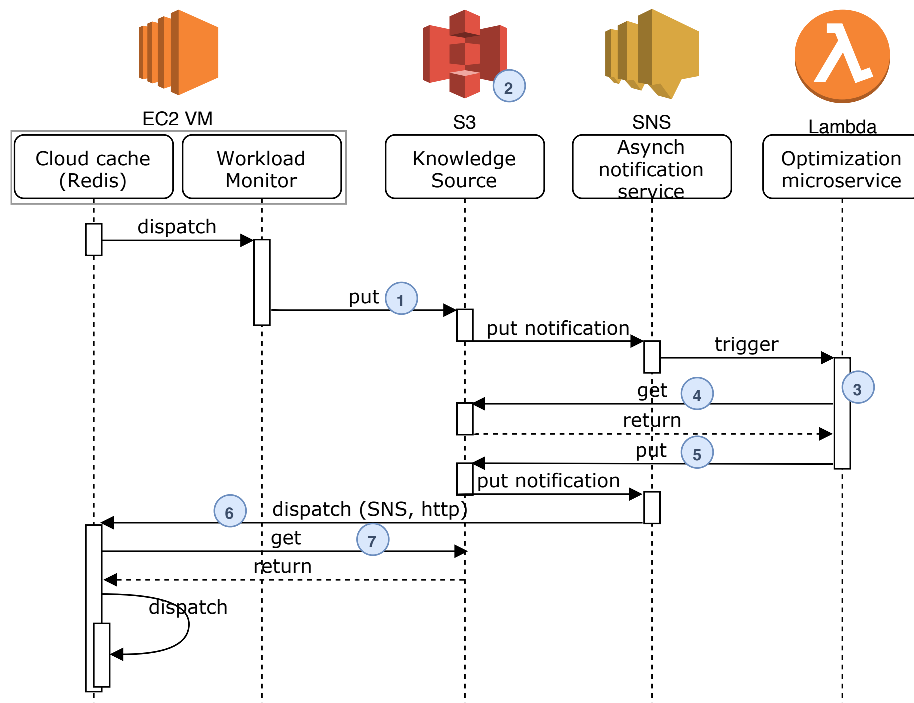

Figure 8.2: Components of SPREDS and their interactions. Table 8.4 describes the AWS services used. The circled numbers in the diagram identify the elements for which AWS charges fees.

To validate our design, we implemented SPREDS as a modified version of the Redis key-value store that leverages several AWS services to periodically self-partition the memory seeking to maximize overall system utility, as depicted in Figure 8.2. The current implementation of SPREDS extends Redis version 4. The Amazon Web Services products used in SPREDS are described in Table 8.4.

Table 8.4: Amazon Web Services Products used in the SPREDS implementation.

Service name and description	**Use in SPREDS**
Elastic Compute Cloud (EC2): Server provisioning	Node hosting cache and monitoring
Simple Storage Service (S3): Serverless object storage	Store the monitoring data and adaptation plan
Lambda: Serverless computing (FaaS)	Serverless adaptation microservice (solver)
Simple Notification Service (SNS): Pub-sub messaging	Delivers notifications when calculate new adaptation and execute adaptation events are triggered

8.4.1 Summary of how SPREDS works and outline for the remainder of the Section

SPREDS captures a very small percentage of the requests it receives. The sampled requests are sent in real time to the workload monitor (Section 8.4.2). The monitor is an external process running on the cache node; it temporarily stores the samples until a configurable number of N samples is received. The monitor then stores the N samples remotely on a S3 bucket. This triggers an SNS event that launches an AWS Lambda function to run the solver, which in turn finds the optimal partitioning of the cache using one of the two solvers (Section 8.4.3). Based on the solution to the optimization problem, the Lambda function generates an adaptation plan to implement the partitioning. This plan is the output of the Lambda function, which it stores on an S3 bucket. Adding a new file to the S3 bucket triggers an *execute adaptation* event which is delivered asynchronously to the cache node using an SNS notification. When the cache receives the adaptation plan, it re-configures itself according to the optimally calculated cache partitions (Section 8.4.4). This loop runs continuously so that the cache can adapt to changes in workload or application demand.

8.4.2 Workload monitor

In earlier work, we instrumented Memcached to construct online miss rate curves (MRCs) [107]. We adopted this same approach in SPREDS, with minor changes due to the difference in internal implementation of Redis versus Memcached. To the best of our knowledge, this is the first implementation of the SHARDS [156] MRC estimation algorithm on Redis. Next, we discuss some of the main challenges in the implementation of the lightweight monitoring approach used in SPREDS.

In computing, caches are used to accelerate access to objects stored in slower memory tiers. For this reason, caches are designed to be extremely fast when answering requests. It is important that any new functionality added to the caching software has minimal or no latency overhead. In our implementation, we move the monitoring of each request out of the critical path by implementing the monitoring agent as an external process and communicate using a high-performant asynchronous inter-process communication library (ZeroMQ[13]).

Another challenge is the trade-off between the sampling frequency and the accuracy of the metrics obtained. We use uniform random spatial sampling [156, 46, 157] to keep track of caching metrics with a low overhead. A function of the hash value of the object determines whether the object should be monitored or not, ensuring that all accesses to the same object are always monitored but only accesses to a small subset of the objects are tracked. The resulting reference stream is a scaled down representative and statistically self-similar version of the original reference stream [156, 28, 157]. Cache metrics can then be scaled up to approximate the metrics of the full reference stream [156, 157]. The overhead introduced by the sampling method is very low for key-value stores, as these already hash the object keys before inserting them. This hashing operation can be used to implement a sampling filter [156] with sampling rate $R = T/P$ using the following operation:

$$hash(key) \bmod P < T \tag{8.7}$$

where T and P are configurable parameters, *hash* is the hashing function used by the key-value store to locate an object based on its user-define *key*. A reference to an object in the cache is sent to the monitoring function if and only if it satisfies the condition in (8.7). Each object that passes the sampling filter condition is monitored and it represents P/T objects in the original object request stream. To generate accurate estimation of the miss rate curves (MRCs) based on this small sample, we use the SHARDS algorithm [156]. As reported by Waldspurger et al. [156], SHARDS can process up to 17M requests per second and build MRCs that are accurate within 2.6% of the original MRC, with a constant memory footprint. SPREDS generates one MRC curve for every cache partition (each of which corresponding to a workload or application).

8.4.3 Optimization (solver)

We use two AWS Lambda functions, one for each of the solvers in SPREDS. AWS Lambda is a Function-as-a-Service offering from Amazon Web Services that lets user run on-demand cloud functions that can be directly invoked or triggered by events such as adding a new file to S3. AWS Lambda charges per function invocation and as a result is typically cheaper than paying for an always-on service running on a dedicated virtual machine or container.

The first Lambda function finds a solution to the optimization problem using Algorithm 8. The

[13] `https://zeromq.org`

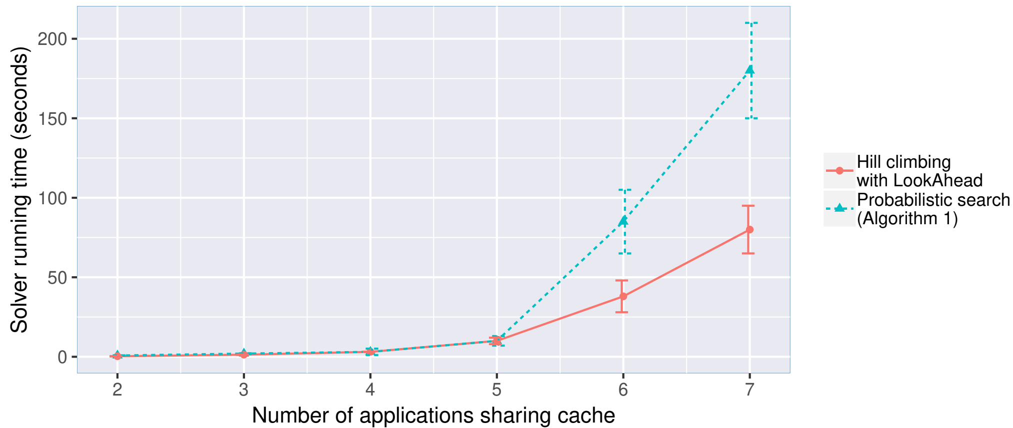

Figure 8.3: Average running time of the two solvers, for a varying number of applications sharing the cache (partitions). Error bars show one standard deviation from the average.

second function finds a solution to the problem using a local search (hill climbing) algorithm. We use the following heuristic to chose between the two solvers: The genetic algorithm is used when the problem is small (≤ 7 applications sharing the cache); otherwise, we use the hill climbing solver. We base this heuristic on observations made on exploratory experimental results (see Figure 8.3): The performance of the genetic solver degrades more rapidly than that of the hill climbing algorithm, as the complexity of the optimization increases. The workloads for each partition were chosen by randomly selecting sub-traces from the Yahoo workload; experiments with other traces yielded similar results.

8.4.4 Adaptation plan and execution

The adaptation plan instructs how to re-partition the cache according to the new solution to the optimization problem. Redis supports online manual re-partitioning through CONFIG SET commands which can be communicated to the cache instances through an HTTP API. We leverage these commands and express the adaptation plan as a set of CONFIG SET commands that change the partition sizes to match the solution found by the solver.

8.5 Experimental validation and cost analysis

As the performance improvements due to re-partitioning the cache based on workload and application demand has already been demonstrated in prior studies [137, 45, 82, 46, 47, 29, 147, 50], we concentrate on the specifics of the design proposed in this chapter. Concretely, we conducted experiments to: (1) confirm that the overhead introduced by the monitoring function is small, (2)

demonstrate the usefulness in a realistic scenario, and (3) quantify the cost savings that can be achieved due to the serverless architecture proposed in this chapter.

Table 8.5: Description of the testbeds used in our experiments. All services used Ubuntu 16.04 LTS as operating system.

Type	Machine characteristics	Software
Local	1 server (8 cores, hyperthreading, 8 GB RAM)	VirtualBox (2 VMs)
	VM1 (4 vCPUs, 4 GB RAM)	YCSB
	VM2 (4 vCPUs, 4 GB RAM)	SPREDS/Redis v4
AWS	t2.medium instance (2 vCPUs, 4 GB RAM)	YCSB
	t2.medium instance (2 vCPUs, 4 GB RAM)	KV-replay [36]
	t2.xlarge instance (4 vCPUs, 16 GB RAM)	SPREDS/Redis v4
	t2.micro instance (1 vCPU, 1 GB RAM)	In-house orchestration scripts

To study the performance overhead of our monitoring approach, we used a local testbed with one server and two virtual machines, one for the cache and one for the workload generator. We also ran a set of cloud (AWS) experiments in which we observe how SPREDS can self-tune and re-partition the cache memory seeking to maximize the overall system utility. For these set of experiments we had two workload generation instances—one for each workload sharing the cache—and a third instance with the cache. In addition, the experiment orchestration scripts were run on a separate micro instance. The details of the machine characteristics and software used are presented in Table 8.5. The storage backends were simulated by introducing an exponentially distributed latency overhead on a cache miss.

8.5.1 Workloads

We used one application workload for the performance overhead experiments (Section 8.5.2) and two different application workloads sharing the same cache node for the usefulness tests (Section 8.5.3). The first workload—used in both sets of experiments—is a classical skewed-popularity workload in which the popularity of keys follow a Zipf distribution. Accesses are generated using the Independent Reference Model (IRM) [8], by sampling from the popularity distribution. The requests are issued using the YCSB's [48] benchmark for cloud and key-value storage systems, configured with the workload C of this benchnmark. For the second workload, we used KV-replay [36] to replay an access trace from a large HDFS[14] deployment at Yahoo [7]. For more details on these workloads, we refer the reader to [48], [7] and [36].

[14]The Hadoop Distributed File System (HDFS) is a distributed file system commonly used when implementing a data lake for analytics of massive datasets.[7]

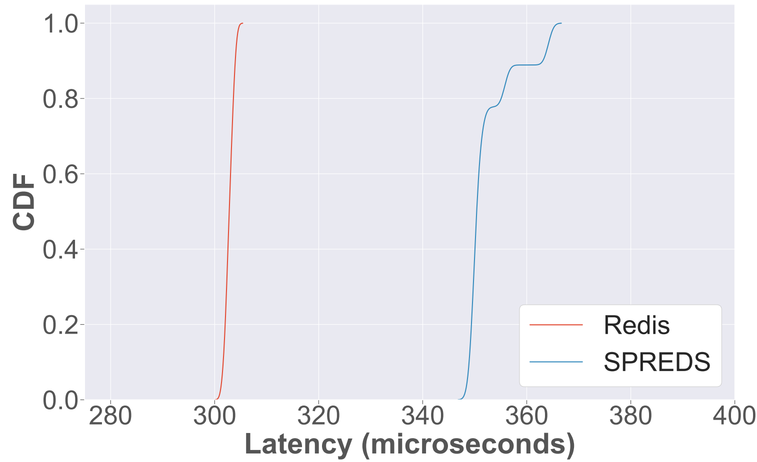

Figure 8.4: Request latency for Redis and SPREDS (with monitoring but without self-tuning). The monitoring in SPREDS introduces a 15% overhead (median latency); however this overhead becomes negligible once the system self-tunes.

8.5.2 Performance overhead

To quantify the overhead introduced by the monitoring mechanism we ran experiments measuring the client-side latency for vanilla Redis and for SPREDS with monitoring but without the self-tuning feedback loop. This latency is the time it takes since a client issues a request until it gets a response back from the cache. The results are shown in Figure 8.4. The performance impact introduced by the monitoring mechanism is small, with a median latency that is 15% slower when the workload is being monitored. However, once this information is used to self-partition the cache, the performance overhead disappears—SPREDS is faster than Redis with static partitioning—as it is offset by the benefits in performance due to the optimized partitioning mechanism (see Figure 8.5).

Additional to the overhead introduced by the monitoring mechanism, there is an overhead that results from storing and reading data from the knowledge source (S3) and from the asynchronous event notifications used in SPREDS (SNS); these steps can be observed in Figure 8.2. We measured this overhead in a fully functional SPREDS implementation (same experiments used in Section 8.5.3), and found that they average ∼1,800ms per adaptation cycle versus ∼100ms for the case of an implementation using an external always-on controller. This slowdown is big (9x) but does not affect overall system performance as the adaptation loop is executed infrequently, with reasonable values being in the order of 1 through 48 cycles per day, so this has a low impact on the performance of the full system.

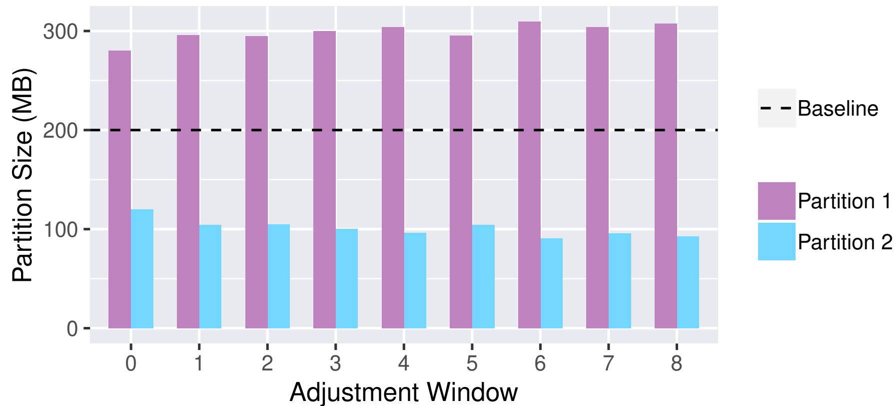

Figure 8.5: Ten-hour sample run of SPREDS with two application workloads (and corresponding cache partitions).

8.5.3 Usefulness

To demonstrate the usefulness of SPREDS, we present the results of a sample run in which SPREDS shows improved performance over naive static partitioning with Redis. For numerous other positive results of optimal cache partitioning in prior related work see Section 8.2.

We ran an experiment on virtual machines using AWS EC2, with a setup in which a cache node is shared between two applications. The cache is partitioned into two Redis instances, one for each workload described earlier in this Section. We ran the experiments five times and present the results of one representative run. The cache was re-partitioned hourly during each ten-hour run, as given by the results of solving the optimization problem. The first application issued 675 K requests during the test, with a skewed popularity distribution (Zipf, YCSB). This application is served by cache instance P1 and a storage backend that has an exponentially distributed request latency ($1/\lambda = 200ms$). The second application issued 61.2 K requests during the test, as replayed from the Yahoo trace. This application is served by cache instance P2 and a storage backend with exponentially distributed latency ($1/\lambda = 800ms$). At the beginning of the experiment, P1 and P2 are statically assigned 200 MB. Figure 8.5 shows how the size of the partitions are adjusted every hour. Table 8.6 quantifies the results with several performance metrics. We can observe that the hit rate of the cache that is accessed more frequently (P1) increases from 54.1% to 61.7%: a 1.42x speedup. This comes at a cost of a lesser decrease in hit rate for P2 (38.6% to 35.2%) and a corresponding slowdown (0.9). As the relative improvement perceived by P1 is greater than the penalty suffered by P2, and as P1 is more frequently accessed than P2, the overall utility improves by 7%.

Table 8.6: Performance improvement of SPREDS versus Redis, after a 10-hour run with 9 self-partitioning cycles, as shown in Figure 8.5.

Metric	Redis	SPREDS
Hit Rate, P1	54.1%	61.7%
Hit Rate, P2	38.6%	35.2%
Effective Access Time, P1	21.6 ms	15.2 ms
Effective Access Time, P2	302.1 ms	335.9 ms
Overall Utility (weighted average of EATs)	44.9 ms	41.9 ms

8.5.4 Costs

We calculate the cost of implementing the autonomic controller as an on-demand serverless computing service. The circled numbers in Figure 8.2 show the interactions and components for which AWS charges. The costs of running the cache itself are not included in the calculations; this cost is the same for all implementations. The monitor runs on the same machine as the cache and does not lead to additional charges. Table 8.7 details what these items represent, as well as the associated costs. The estimate given for the knowledge storage in S3 is for a 6-month data retention policy. Note that some of the services listed in the table contain free service provisions: Lambda (1 million requests per month and 400,000 GB-seconds of compute time per month), SNS (1 million publishes per month, 100 thousand notifications per month, and all notifications to Lambda), and S3 (all deletes and data transfers between S3 and VMs or Lambdas in the same region). The free tier provisions that we are considering are those that are available to all AWS clients; these are unlike the EC2 free micro instances, which are only free for 12 months for new clients.[15]

The costs detailed in Table 8.7 are for one adaptation cycle. For the problem studied in this chapter, the reasonable number of adaptations per day ranges from one cycle per day to 96 per day—in other words, one adaptation a day to one adaptation every 15 minutes.

Table 8.7: AWS costs, for items enumerated in Figure 8.2 (as of December 2019, AWS us-east-2 region).

Item	Description	Cost	Quantity
1,5	S3 PUT	$0.005 per 1,000 requests	Two PUTs
4,7	S3 GET	$0.0004 per 1,000 requests	Two GETs
2	S3	$0.023 per GB/month	280 MB/month
3	Lambda	$0.000001667 per 100ms; 1GB RAM	A 1 min run
6	http SNS	$0.60 per million	1

Figure 8.6 shows that, when the adaptation loop is executed hourly, the **yearly** costs of running the controller as a serverless microservice is $1.92 or 0.85% of the cost of the always-on service—if the tenant does not exceed the AWS free service provisions. If these are exceeded by other

[15]For more details, see: `https://aws.amazon.com/free`

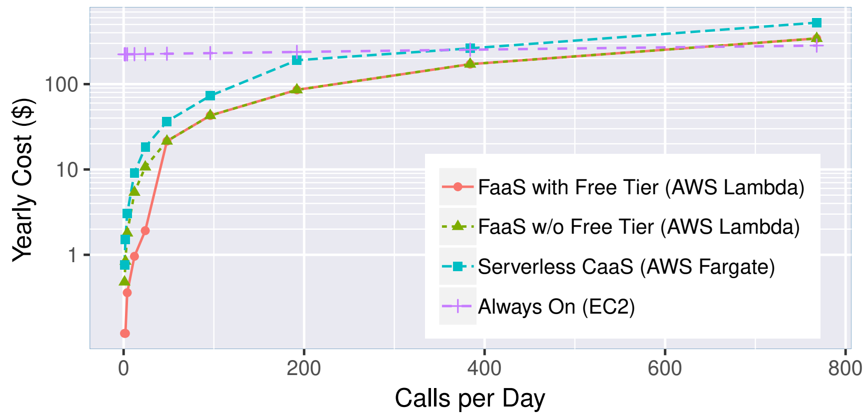

Figure 8.6: Costs of running the autonomic controller on AWS as an always-on service, and as an on-demand serverless microservice (using FaaS and CaaS services). The y-axis is in log scale.

tenant activities, then the yearly cost is $10.13 or less than 5% of the cost of the always-on service. These numbers are for a 1-minute optimization solver, which we calculated is enough to solve the optimizations that arise in a normal Redis deployment (for example, Redis documentation suggests 6 partitions in its tutorials). For more complex problems, the adaptation loop is likely to be run less frequently and for many configurations expected in production, the serverless approach is expected to be better (in terms of cost) than the always-on approach.

The Figure also shows the costs using a serverless Container-as-a-Service (CaaS) product called AWS Fargate.[16] For the case of a 1-minute run, using Lambda is cheaper than using Fargate. However, Fargate becomes an option when the solver running time exceeds the duration limit of the FaaS service (e.g., 15 minutes).

Finally, in larger organizations the solver service may be shared by multiple applications and the always-on service may be a better alternative. For example—for the specific scenario described in this Section—the always-on approach becomes a better option when the service is called 606 or more times per day (see Figure 8.6). At one call per hour, an organization should have at least 26 systems using the shared service for the always-on service to be the best approach. Properly choosing the deployment option—always-on serverless functions or serverless containers—is a challenge to be addressed in real installations.

8.6 Open challenges

We argued for a serverless computing approach for the adaptation component of a self-optimizing system. Our proof-of-concept implementation and cost analysis show that this is feasible and useful.

[16]We show only one cost curve for Fargate as the lines with and without free tier provisioning overlap for this service. There are no free Fargate invocations; only SNS notifications and some calls to S3 are free in this scenario.

In this section we identify four challenges that must be addressed for this vision to become a reality.

First, cloud functions have a maximum running time (AWS: 15min, Azure: 10min, Google: 9min). If the calculations exceed the limit, the function expires and the work is lost. While it is possible for the limit to be extended in the near future,[17] it is nevertheless possible that some problems take too long to solve as cloud functions. In the AWS ecosystem, the Fargate serverless container service can be used for such problems. The challenge is to design a system that can **automatically** use the cheapest service for its specific size (in terms of partitions) and frequency of its adaptation loop.

Second, some systems for which a serverless approach makes sense, may outgrow it and require an always-on service. Again, a cloud-native solution should automatically choose the architecture that is best for the specific system configuration and complexity. This adaptation should be dynamic and transparent.

Third, we identify a challenge not specific to serverless architectures: interacting self-aware applications that use information from other systems during the adaptation decisions [92]. As the serverless approach decouples the adaptation components from the main systems, such a system could be orchestrated if we provide proper APIs and connectors.

Fourth, even though we focused on adaptation decisions (specially those related to performance tuning), the community should think about realizing the vision of fully self-aware systems [92]. Could a cloud-native architecture that leverages serverless computing offerings help in realizing such systems?

8.7 Concluding remarks

The idea that complex software systems should be autonomic and self-adaptive has been around for more than a decade. Yet, while some adaptive functionality has made it to commercial systems, most real databases still depend on manual (expert) tuning. One reason self-adaptation mechanisms have not made it into production is that their tuning tends to require exploring a large configuration space or running expensive algorithms—prohibitive operations that compete for the valuable and limited resources (CPU and memory) of the system being tuned. A way to overcome the limitations described above is to adopt a serverless microservice design for the controller. Through SPREDS, a proof-of-concept self-partitioning cloud cache, we demonstrate that this approach is feasible, useful, and overcomes the cost and performance limitations of prior designs. We encourage the community to adopt this architectural design, so that cloud-based autonomic systems can be affordably implemented in production.

[17]For example, in 2018 AWS increased its Lambda limit from 5 to 15 min. Source: `https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/`

Chapter 9

Conclusions and future directions

9.1 Conclusions

In this dissertation, we studied the problem of using modern cloud platforms to deploy applications based on a microservices architecture. Our focus is on the on-demand cloud computing platforms, like containers and serverless, where an environment initialization phase is required to allow the execution of the microservices. This initialization phase often increase the latency of the deployed applications. Consequently, a higher latency makes it difficult to meet expected service levels (SLOs) required by applications that are sensitive to response times, preventing their migration towards microservices architectures executed on cloud platforms.

We began our work with the serverless platform, specifically the Function-as-a-Services platforms. We found that current task scheduling algorithms are plain load-balancers, and we proposed 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, showed that the latency of cloud functions can be effectively reduced.

We then implemented our proposed scheduling algorithm in the OpenLambda FaaS platform, to validate the feasibility of the algorithm wich leads to better cloud function latency, while keeping a low level of node unbalance. To deepen the analysis of the impact of using code locality as part of the scheduling process, we carried out an evaluation with real workloads and compared the proposed algorithm with other state-of-the-art scheduling algorithms. These experiments showed that relaxing the load-balancing requirement and changing it to a less restrictive goal gives us more flexibility in mapping decisions that can lead to considerable performance gains by exploiting code locality.

We also studied how microservices are deployed using containers. In this platforms, an orchestration system is required to manage the complexity of the deployment, including scaling of the deployed services. Our analysis showed that the default scale-out deployment mechanisms are not performance-aware, leading to poor performance of the microservices. We also found that placement

decisions are critical for both initialization time and run-time performance of the containers, and proposed a closed-loop system that helps to maintain the SLOs by scaling or migrating containers based on the resource variability in cloud environments.

To extend the analysis of the impact of smart scheduling decisions to other cloud components required for the deployment of microservices-based application, we studied the case of data lakes. We proposed the use of algorithms that prioritize maximizing the use of the cache, without neglecting load balancing, as a method to obtain better access times to the data stored in these systems. Our work shows that using cache affinity policies in the caching layer for data lake systems is better than using a more common load balancing policy.

Finally, we made a case for the use of serverless computing cloud services to solve the complex optimization problems that arise during self-tuning of cloud components, like cloud caches. We implemented SPREDS (Self-Partitioning REDiS), a modified version of Redis that leverages modern data structures and statistical sampling methods to efficiently obtain online estimates of the real miss rate curves and optimizes memory management in the multi-instance Redis scenario. A cost analysis shows that the serverless computing approach can lead to significant cost savings.

9.2 Future directions

The results obtained in this research leads to the surge of new research questions that can be addressed in the future:

- Could delaying microservice launch time, when they are part of complex workflows, help improve performance?

  Prioritization is a well known way to improve the performance of tasks scheduling algorithms. In the case of microservices, if they are part of a long or complex workflow, it could be beneficial to use the expected execution order as a priority weight for the scheduling algorithm. Delaying microservices with lower priority will allow to reduce the pressure in the system, leading to a faster initialization of those tasks with higher priority (i.e. execution order).

- Could the provider offer differentiated services so that we only need to optimize the scheduling of those microservices that need quick launch time?

  A differentiated service applied to the workloads for on-demand cloud computing platforms, can be used as a form of prioritization, allowing the cloud platform to provide more or better resources to a reduced amount of microservices, which could lead to a reduced latency.

- Is it possible to build a model for the impact of worker's available CPU, on the performance of container-based microservices?

  Our work on containers platform showed the existence of a relation between the CPU load and the performance of the microservices. However, a model that allows to predict the impact

of the current CPU load, before the container is placed could lead to a mayor improvement in the scheduling algorithm. For example, if the model shows that a specific new container will perform similarly on both a node with less CPU capacity and a node with a higher CPU capacity, it would be more convenient to place the container in the node with less CPU capacity, to allow more sensitive containers to run in the less loaded node.

Bibliography

[1] Cristina L. Abad, Andres G. Abad, and Luis E. Lucio. Dynamic memory partitioning for cloud caches with heterogeneous backends. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 87–90, 2017.

[2] Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. Package-Aware Scheduling of FaaS Functions. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion*. ACM, 2018.

[3] Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. Package-aware scheduling of faas functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 101–106, 2018.

[4] Cristina L. Abad, Alexandru Iosup, Edwin F. Boza, and Eduardo Ortiz Holguin. An analysis of distributed systems syllabi with a focus on performance-related topics. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 121–126, 2021.

[5] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. Metadata traces and workload models for evaluating big storage systems. In *2012 ieee fifth international conference on utility and cloud computing*, pages 125–132. IEEE, 2012.

[6] Cristina L. Abad, Eduardo Ortiz-Holguin, and Edwin F. Boza. Have we reached consensus? an analysis of distributed systems syllabi. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 1082–1088, 2021.

[7] Cristina L. Abad, Nathan Roberts, Yi Lu, and Roy H. Campbell. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 100–109, 2012.

[8] Cristina L. Abad, Mindi Yuan, Chris X. Cai, Yi Lu, Nathan Roberts, and Roy H. Campbell. Generating request streams on Big Data using clustered renewal processes. *Performance Evaluation*, 70(10), 2013.

[9] Cristina Abad Robalino, Mónica Villavicencio Cabezas, Edwin Boza Gaibor, and Glenda Ortega Campuzano. *Informe del estado de adopción de tecnologías de computación en la nube en el Ecuador. Una encuesta del Grupo de Investigación en Big Data de la ESPOL*. Escuela Superior Politécnica del Litoral, 1 edition, 2018.

[10] Mania Abdi, Amin Mosayyebzadeh, Mohammad Hossein Hajkazemi, Emine Ugur Kaynar, Ata Turk, Larry Rudolph, Orran Krieger, and Peter Desnoyers. A community cache with complete information. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pages 323–340, 2021.

[11] Gojko Adzic and Robert Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 884–889. ACM, 2017.

[12] Sascha Alpers, Christoph Becker, Andreas Oberweis, and Thomas Schuster. Microservice based tool support for business process modelling. In *Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International*, pages 71–78. IEEE, 2015.

[13] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015.

[14] George Anadiotis. Zdnet article — a rock and a hard place: Between scylladb and cassandra, 2017. [Online] Available at: `https://www.zdnet.com/article/a-rock-and-a-hard-place-between-scylladb-and-cassandra/`. Last accessed: April 6, 2021.

[15] Xavier Andrade, Jorge Cedeno, Edwin F. Boza, Harold Aragon, Cristina L. Abad, and Jorge Murillo. Optimizing cloud caches for free: A case for autonomic systems with a serverless computing approach. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 140–145. IEEE, 2019.

[16] Harold Aragon, Samuel Braganza, Edwin F. Boza, Jonathan Parrales, and Cristina L. Abad. Workload Characterization of a Software-as-a-Service Web Application Implemented with a Microservices Architecture. In *Companion Proceedings of The 2019 World Wide Web Conference*, pages 746–750, 2019.

[17] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, April 2010.

[18] William C Arnold, DJ Arroyo, Wolfgang Segmuller, Mike Spreitzer, Malgorzata Steinder, and Asser N Tantawi. Workload orchestration and optimization for software defined environments. *IBM Journal of Research and Development*, 58(2/3), 2014.

[19] Matt Asay. Why Kubernetes is winning the container war, September 2016.

[20] Gabriel Aumala, Edwin F. Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina L. Abad. Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 282–291, 2019.

[21] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[22] IBM Autonomic Computing. An architectural blueprint for autonomic computing. *IBM White Paper, Fourth Edition*, 2006.

[23] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[24] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and others. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[25] Davide B Bartolini, Filippo Sironi, Donatella Sciuto, and Marco D Santambrogio. Automated fine-grained cpu provisioning for virtual machines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):1–25, 2014.

[26] André Bauer, Veronika Lesch, Laurens Versluis, Alexey Ilyushkin, Nikolas Herbst, and Samuel Kounev. Chamulteon: Coordinated auto-scaling of micro-services. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2015–2025. IEEE, 2019.

[27] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, and Yves Robert. Centralized versus distributed schedulers for bag-of-tasks applications. *IEEE Transactions on Parallel and Distributed Systems*, 19(5), 2008.

[28] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75. IEEE, 2015.

[29] Daniel Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching. In *USENIX OSDI*, 2018.

[30] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128. IEEE, 2007.

[31] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650. USENIX Association, 2018.

[32] Edwin F. Boza, Cristina L. Abad, Shankaranarayanan Puzhavakath Narayanan, Bharath Balasubramanian, and Minsung Jang. A Case for Performance-Aware Deployment of Containers. In *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, pages 25–30, 2019.

[33] Edwin F. Boza, Cristina L. Abad, Mónica Villavicencio, Stephany Quimba, and Juan Antonio Plaza. Reserved, on demand or serverless: Model-based simulations for cloud budget planning. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, October 2017.

[34] Edwin F. Boza, Cristina L. Abad, Mónica Villavicencio, and Stephany Quimba. Cloud Adoption in Ecuador: Current Status, Challenges and Opportunities. In *2018 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6. IEEE, 2018.

[35] Edwin F. Boza, Xavier Andrade, Jorge Cedeno, Jorge Murillo, Harold Aragon, Cristina L. Abad, and Andres G. Abad. On Implementing Autonomic Systems with a Serverless Computing Approach: The Case of Self-Partitioning Cloud Caches. *Computers*, 9(1):14, 2020.

[36] Edwin F. Boza, Cesar San-Lucas, Cristina L. Abad, and Jose A. Viteri. Benchmarking key-value stores via trace replay. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 183–189. IEEE, 2017.

[37] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, 2016.

[38] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pages 1–6. IEEE, 2016.

[39] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco AS

Netto, and others. A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade. *arXiv preprint arXiv:1711.09123*, 2017.

[40] Phil Calçado. *How we ended up with microservices.* Sep, 2015.

[41] Marco Capuccini, Anders Larsson, Salman Toor, and Ola Spjuth. KubeNow: A Cloud Agnostic Platform for Microservice-Oriented Applications. In *OASIcs-OpenAccess Series in Informatics*, volume 60. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[42] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002.

[43] Ludmila Cherkasova and Shankar R Ponnekanti. Optimizing a content-aware load balancing strategy for shared web hosting service. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No. PR00728)*, pages 492–499. IEEE, 2000.

[44] Andrew Chung, Jun Woo Park, and Gregory R Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 121–134, 2018.

[45] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *USENIX HotCloud*, 2015.

[46] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, 2016.

[47] Asaf Cidon, Daniel Rushton, Stephen Rumble, and Ryan Stutsman. Memshare: A dynamic multi-tenant key-value cache. In *Usenix ATC*, 2017.

[48] Brian Cooper, Adam Silberstein, Erwin Tam, Ramakrishnan, and Sears. Benchmarking cloud serving systems with YCSB. In *ACM SOCC*, 2010.

[49] Daniele D'Agostino, Luca Roverelli, Gabriele Zereik, Andrea De Luca, Ruben Salvaterra, Andrea Belfiore, Gianni Lisini, Giovanni Novara, and Andrea Tiengo. A microservice-based portal for X-ray transient and variable sources. *PeerJ Preprints*, 5:e2519, 2017.

[50] Sudipto Das, Miroslav Grbic, et al. Automatically indexing millions of databases in microsoft azure sql database. In *ACM SIGMOD*, 2019.

[51] DB-engines ranking of key-value stores. [Online] Available at: `db-engines.com/en/ranking/key-value+store`. Accessed: December 31, 2019.

[52] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[53] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Notices (ASPLOS)*, 48(4), 2013.

[54] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: trends, focus, and potential for industrial adoption. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 21–30. IEEE, 2017.

[55] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: How to make your application scale. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 95–104. Springer, 2017.

[56] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, 2019.

[57] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3), 2016.

[58] Gil Einziger and Roy Friedman. TinyLFU: A highly efficient cache admission policy. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2014.

[59] Payam Emami Khoonsari, Pablo Moreno, Sven Bergmann, Joachim Burman, Marco Capuccini, Matteo Carone, Marta Cascante, Pedro de Atauri, Carles Foguet, Alejandra N Gonzalez-Beltran, and others. Interoperable and scalable data analysis with microservices: Applications in metabolomics. *Bioinformatics*, 35(19):3752–3760, 2019. Publisher: Oxford University Press.

[60] Huang Fang. Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem. In *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, pages 820–824. IEEE, 2015.

[61] Maria Fazio, Antonio Celesti, Rajiv Ranjan, Chang Liu, Lydia Chen, and Massimo Villari. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, 3(5):81–88, 2016.

[62] Dror Feitelson. Job scheduling in multiprogrammed parallel systems. Technical report, IBM Thomas J. Watson Research Center, 1997. IBM Research Report 19790.

[63] José Bravo Ferreira, Marco Cello, and Jesús Omana Iglesias. More sharing, more benefits? a study of library sharing in container-based infrastructures. In *European Conference on Parallel Processing*, pages 358–371. Springer, 2017.

[64] Martin Fowler and James Lewis. Microservices, 2014. *URL: http://martinfowler.com/articles/microservices.html*, 1(1):1–1, 2014.

[65] Michael Fredman and Robert Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3), July 1987.

[66] Yu Gan and Christina Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.

[67] Yu Gan, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Yanqi Zhang, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, Christina Delimitrou, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, and Brian Ritchken. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19*, pages 3–18, Providence, RI, USA, 2019. ACM Press.

[68] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 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 - ASPLOS '19*, pages 19–33, Providence, RI, USA, 2019. ACM Press.

[69] Panagiotis Garefalakis, Konstantinos Karanasos, Peter R Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *EuroSys*, pages 4–1, 2018.

[70] Simson Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3, and sqs. *Harvard Computer Science Group Technical Report TR-08-07*, 2007.

[71] Sona Ghahremani, Holger Giese, and Thomas Vogel. Efficient utility-driven self-healing employing adaptation rules for large dynamic architectures. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 59–68. IEEE, 2017.

[72] Jim Gray. A conversation with Werner Vogels. *ACM Queue*, 4(4):14–22, 2006.

[73] Lin Gu, Junjian Guan, Song Wu, Hai Jin, Jia Rao, Kun Suo, and Deze Zeng. Cntc: A container aware network traffic control framework. In *International Conference on Green, Pervasive, and Cloud Computing*, pages 208–222. Springer, 2019.

[74] Carlos Guerrero, Isaac Lera, and Carlos Juiz. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing*, 16(1):113–135, 2018.

[75] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. Analysis of Join-the-Shortest-Queue Routing for Web server farms. *Performance Evaluation*, 64(9-12):1062–1081, 2007.

[76] Aurelien Havet, Valerio Schiavoni, Pascal Felber, Maxime Colmant, Romain Rouvoy, and Christof Fetzer. Genpack: A generational scheduler for cloud data centers. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 95–104. IEEE, 2017.

[77] Parisa Heidari and Ali Kanso. Qos assurance through low level analysis of resource utilization of the cloud applications. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 228–235. IEEE, 2016.

[78] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 223–226, 2017.

[79] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[80] Nikolas Herbst, André Bauer, Samuel Kounev, Giorgos Oikonomou, Erwin van Eyk, George Kousiouris, Athanasia Evangelinou, Rouven Krebs, Tim Brecht, Cristina L. Abad, et al. Quantifying cloud performance and dependability: Taxonomy, metric design, and emerging challenges. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)*, 3(4):1–36, 2018.

[81] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[82] Xiameng Hu, Xiaolin Wang, Yechen Li, et al. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Usenix ATC*, 2015.

[83] Kai Huang, Ke Wang, Dandan Zheng, Xiaoxu Zhang, and Xiaolang Yan. Access adaptive and thread-aware cache partitioning in multicore systems. *MDPI Electronics*, 7(9), 2018.

[84] Stratos Idreos, Niv Dayan, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[85] Yichen Jia, Zili Shao, and Feng Chen. Slimcache: Exploiting data compression opportunities in flash-based key-value caching. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 209–222. IEEE, 2018.

[86] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32, 2019.

[87] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, and others. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, page 33. ACM, 2018.

[88] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Comp. Netw.*, 31(11), 1999.

[89] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating Network-based CPU in Container Environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[90] Kevin Khanda, Dilshat Salikhov, Kamill Gusmanov, Manuel Mazzara, and Nikolaos Mavridis. Microservice-based iot for smart buildings. In *Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on*, pages 302–308. IEEE, 2017.

[91] Hamzeh Khazaei, Alireza Ghanbari, and Marin Litoiu. Adaptation as a service. In *ACM CASCON*, 2018.

[92] Samuel Kounev, Peter Lewis, Kirstie Bellman, Bencomo, et al. The notion of self-aware computing. In *Self-Aware Computing Systems*. Springer, 2017.

[93] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. Designing a smart city internet of things platform with microservice architecture. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, pages 25–30. IEEE, 2015.

[94] Khalid Lafi. Package consistent: A Golang implementation of consistent hashing and consistent hashing with bounded loads. Available at: `https://github.com/lafikl/consistent`. Last accessed on Nov. 1st, 2018.

[95] James Larisch, James Mickens, and Eddie Kohler. Alto: lightweight vms using virtualization-aware managed runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–7, 2018.

[96] Youhuizi Li, Jiancheng Zhang, Congfeng Jiang, Jian Wan, and Zujie Ren. Pine: Optimizing performance isolation in container environments. *IEEE Access*, 7:30410–30422, 2019.

[97] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169, Orlando, FL, April 2018. IEEE.

[98] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James Larus, and Albert Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable Web services. *Perform. Eval.*, 68(11), 2011.

[99] Ying Lu, Avneesh Saxena, and Tarek Abdelzaher. Differentiated caching services; a control-theoretical approach. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[100] Ying Mao, Jenna Oak, Anthony Pompili, Daniel Beer, Tao Han, and Peizhao Hu. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International*, pages 1–8. IEEE, 2017.

[101] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2), 1970.

[102] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 257–262, 2016.

[103] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[104] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 587–604. SIAM, 2018.

[105] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[106] Jeffrey C Mogul and Ramana Rao Kompella. Inferring the Network Latency Requirements of Cloud Tenants. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[107] Jorge R Murillo, Gustavo Totoy, and Cristina L Abad. Instrumenting cloud caches for online workload monitoring: The case of online miss rate curve estimation in memcached. In *Proceedings of the 16th Workshop on Adaptive and Reflective Middleware*, pages 1–6, 2017.

[108] Muhammad Nasir, Hiroshi Horii, Marco Serafini, Nicolas Kourtellis, Rudy Raymond, Sarunas Girdzijauskas, and Takayuki Osogami. Load balancing for skewed streams on heterogeneous cluster. *arXiv preprint arXiv:1705.09073*, 2017.

[109] Muhammad Nasir, Gianmarco Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *IEEE Intl. Conf. Data Eng. (ICDE)*, 2015.

[110] Using NGINX as HTTP load balancer. Available at: `http://nginx.org/en/docs/http/load_balancing.html`. Last accessed: Dec. 13, 2018.

[111] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Pipsqueak: Lean Lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400. IEEE, 2017.

[112] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Pipsqueak: Lean Lambdas with large libraries, 2017. (Presentation at) Intl. Workshop on Serverless Comp. (WoSC).

[113] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX ATC*, 2018.

[114] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[115] Claus Pahl. Containerisation and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[116] Vivek Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. *SIGOPS Oper. Syst. Rev.*, 32(5), 1998.

[117] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. pRedis: Penalty and locality aware memory allocation in redis. In *ACM Symposium on Cloud Computing (SoCC)*, 2019.

[118] Vladimir Podolskiy, Michael Mayo, Abigail Koay, Michael Gerndt, and Panos Patros. Maintaining slos of cloud-native applications via self-adaptive resource sharing. In *2019 IEEE 13th*

International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pages 72–81. IEEE, 2019.

[119] Adam Prügel-Bennett. When a genetic algorithm outperforms hill-climbing. *Theoretical Computer Science*, 320(1), 2004.

[120] Konstantinos Psounis and Balaji Prabhakar. A randomized web-cache replacement scheme. In *IEEE Conf. Comp. Comm. (INFOCOM)*, 2001.

[121] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-optimal, fair cache sharing. In *USENIX NSDI*, 2016.

[122] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning. In *IEEE/ACM MICRO*, 2006.

[123] Joy Rahman and Palden Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210. IEEE, 2019.

[124] Franck Ravat and Yan Zhao. Data lakes: Trends and perspectives. In *International Conference on Database and Expert Systems Applications*, pages 304–313. Springer, 2019.

[125] Dimensional Research. Global microservices trends. Technical report, A Survey of Development Professionals, 2018.

[126] Christophe Restif, Natalia Ponomareva, and Krzysztof Ostrowski. A classifier for the latency-cpu behaviors of serving jobs in distributed environments. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 123–130. IEEE, 2015.

[127] Andrew Roland. Improving load balancing with a new consistent-hashing algorithm, December 2016. Vimeo Engineering Blog, Medium. Available at: `https://link.medium.com/EaD2ZaHAAS`. Last accessed: Dec. 11, 2018.

[128] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[129] Sidecar pattern, 2017. Microsoft Azure online documentation; available at: `https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar`. Last accessed: Feb 13, 2019.

[130] Matthew NO Sadiku, Sarhan M Musa, and Omonowo D Momoh. Cloud computing: opportunities and challenges. *IEEE potentials*, 33(1):34–36, 2014.

[131] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. Data-driven serverless functions for object storage. In *ACM/IFIP/USENIX Middleware*, 2017.

[132] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–135, 2019.

[133] Randy Shoup. From the Monolith to Microservices: Lessons from Google and eBay. In *presentation at Craft Conf*, volume 2016, 2016.

[134] Ankit Singla, Balakrishnan Chandrasekaran, B Godfrey, and Bruce Maggs. The internet at the speed of light. In *ACM HotNets*, 2014.

[135] Ankit Singla, Balakrishnan Chandrasekaran, P Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2014.

[136] Willian Stallings. *Operating Systems Internals And Design Principles, 9th Edition*. Pearson Education, 2018.

[137] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181, 2015.

[138] Ion Stoica, Robert Morris, David Karger, M. Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4), August 2001.

[139] Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1), 2004.

[140] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th international conference on World Wide Web*, pages 331–340, 2007.

[141] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. CNTR: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[142] Dominique Thiébaut, Harold Stone, and Joel Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), 1992.

[143] Stefan Tilkov. The modern cloud-based platform. *IEEE Software*, 32(2):116–116, 2015.

[144] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Jrnl. Par. Distr. Comp. (JPDC)*, 24(2), 1995.

[145] Gustavo Totoy, Edwin F. Boza, and Cristina L. Abad. An Extensible Scheduler for the OpenLambda FaaS Platform. In *2nd Workshop on Hardware/Software Techniques for Minimizing Data Movement (Min-Move 2018)*, 2018.

[146] Eddy Truyen, Dimitri van Landuyt, Bert Lagaisse, and Wouter Joosen. Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 156–159, 2019.

[147] Dana van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *ACM International Conference on Management of Data (SIGMOD)*, 2017.

[148] Erwin van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. The spec-rg reference architecture for faas: From microservices and containers to serverless platforms. *IEEE Internet Computing*, 23(6):7–18, 2019.

[149] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. A SPEC RG cloud group's vision on the performance challenges of FaaS cloud architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 21–24. ACM, 2018.

[150] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The SPEC cloud group's research vision on FaaS and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*, pages 1–4, Las Vegas, Nevada, 2017. ACM Press.

[151] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[152] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, page 18. ACM, 2015.

[153] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *ACM/IFIP/USENIX international conference on distributed systems platforms and open distributed processing*, pages 243–264. Springer, 2008.

[154] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE, 2018.

[155] Jóakim von Kistowski, Nikolas Roman Herbst, and Samuel Kounev. Modeling variations in load intensity over time. In *Proceedings of the third international workshop on Large scale testing*, pages 1–4, 2014.

[156] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient {MRC} construction with {SHARDS}. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

[157] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, 2017.

[158] Weina Wang, Kai Zhu, Lei Ying, Jiang Tan, and Li Zhang. MapTask scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality. *IEEE/ACM Trans. Netw.*, 24(1), 2016.

[159] Sage A. Weil. *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, University of California, Santa Cruz, 2007.

[160] Chen Wu, Rodrigo Tobar, Kevin Vinsen, Andreas Wicenec, Dave Pallot, Baoqiang Lao, Ruonan Wang, Tao An, Mark Boulton, Ian Cooper, and others. DALiuGE: A graph execution framework for harnessing the astronomical data deluge. *Astronomy and Computing*, 20:1–15, 2017.

[161] Miguel G. Xavier, Israel C. De Oliveira, Fabio D. Rossi, Robson D. Dos Passos, Kassiano J. Matteussi, and César AF De Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 253–260. IEEE, 2015.

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

[163] Qiaomin Xie, Mayank Pundir, Yi Lu, Cristina Abad, and Roy Campbell. Pandas: Robust locality-aware scheduling with stochastic delay optimality. *IEEE/ACM Trans. Netw.*, 25(2), 2017.

[164] Cong Xu, Karthick Rajamani, and Wesley Felter. Nbwguard: Realizing network qos for kubernetes. In *Proceedings of the 19th International Middleware Conference Industry*, pages 32–38, 2018.

[165] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dCat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *European Conference on Computer Systems (EuroSys)*, 2018.

[166] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, page 5. ACM, 2016.

[167] Yinghao Yu, Wei Wang, Renfei Huang, Jun Zhang, and Khaled Ben Letaief. Achieving load-balanced, redundancy-free cluster caching with selective partition. *IEEE Transactions on Parallel and Distributed Systems*, 31(2):439–454, 2019.

[168] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *European Conf. Comp. Sys. (EuroSys)*, 2010.