CONTAINER MANAGEMENT FOR SERVERLESS EDGE COMPUTING OFFERINGS

A Thesis

by

CHIH-PENG WU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Dilma Da Silva
Committee Members,	Riccardo Bettati
	Natarajan Gautam
Head of Department,	Scott Schaefer

December 2019

Major Subject: Computer Science

ABSTRACT

Under the serverless paradigm, containers may serve as the runtime execution environments for processing clients' service requests. For service providers aiming at broad customer bases, the portfolio of containers to be made available can be quite large. In edge computing scenarios, where hardware elasticity is limited or nonexistent, an effective method for container provisioning and destroying is crucial to increase service availability and mitigate startup overheads. However, current methods have not been designed for the Internet-of-Things (IoT) applications – one major use case in edge computing.

In this work, we introduce a new container management method that exploits predictable patterns present in the workload to decrease request latency in such environments. We propose a new container management method, called Look-Ahead Request Serving (LARS), designed for IoT applications that exhibit periodicity. We demonstrate that for workloads that invoke requests periodically (e.g., environmental sensors, surveillance cameras, smart home gadgets), our method outperforms the method in OpenWhisk, an open-source serverless platform, attaining a 37% and 78% improvement in the startup overhead in a smart gym and a smart home scenario, respectively.

TABLE OF CONTENTS

Page

ABSTRACT ii

TABLE OF CONTENTS iii

LIST OF FIGURES v

LIST OF TABLES vi

1. INTRODUCTION 1

2. BACKGROUND 4

2.1 Containers 4
2.2 Serverless computing 6
2.3 Edge computing 7

3. PROBLEM DESCRIPTION 9

3.1 Coldstart overhead 9
3.2 State-of-the-art in open-source container management 9
3.3 Terminology 10
3.4 Container management problem 11

4. METHOD 13

4.1 Request serving order 13
4.2 Our method 15
4.3 Reducing complexity by leveraging workload characteristics 20

5. EVALUATION 22

5.1 Testbed and Configurations 22
5.2 Handcrafted workload experiment 23
5.3 Gym scenario 25
5.4 Smart Building Scenario 26
5.5 Reduce the complexity by leveraging workload characteristics 28

6. DISCUSSION AND CONCLUSION 31

6.1 Considering memory size in LARS 31
6.2 Considering more function types and requests in the sub-problem 31

6.3 Caching decision-making for the collections of function requests 31
6.4 Replace average waiting time by other objectives 32
6.5 Conclusion 32

REFERENCES 33

LIST OF FIGURES

FIGURE Page

4.1 Toy example of requests 14

4.2 Order 1 of the toy example 14

4.3 Order 2 of the toy example 15

4.4 Total time breakdown of serving order aabb 17

4.5 Total time breakdown of serving order abab 18

5.1 Ratio versus improvement rate 24

5.2 Number of functions over capacity versus improvement rate. 25

5.3 Improvement rate versus number of users in the gym scenario 27

5.4 Improvement rate versus number of rooms in the building scenario 29

5.5 The compare function reused rate over different number of function 30

LIST OF TABLES

TABLE Page

4.1 Waiting time of different request serving order 19

5.1 Device setup for gym scenario 27

5.2 Device setup for Building scenario 28

1. INTRODUCTION

The emerging serverless computing (or function-as-a-service, FaaS) paradigm [1] is attractive to application developments for several reasons. Developers can focus on the required functionality (building 'functions') while service providers are responsible for taking care of infrastructure tasks such as machine maintenance, security patching, networking configuration and capacity planning [2]. It is the service provider's responsibility to provision the appropriate runtime environment (referred to as *container instances*) to fulfill customer's function requests. Most major cloud service providers offer proprietary serverless platforms, such as AWS Lambda [3], Google Cloud Functions [4], and Microsoft Azure Functions [5]. Furthermore, there are several open-source projects: IBM OpenWhisk [6], OpenLambda [7], OpenFaaS [8], Kubeless [9] and others [10].

In the context of edge computing environments, cloud vendors also offer AWS Greengrass [11] and Microsoft IoT Edge [12] as a way of bringing capabilities to devices in Internet-of-Things (IoT) applications. They aim at offering local computing, messaging, data caching and syncing, and machine language (ML) inference to edge devices.

The FaaS approach is particularly suitable for IoT applications, as they often include devices that sense or generate data to be manipulated through application-specific functions that are too demanding to be carried out by the device itself. The application developer can specify the functions to be executed by the service provider, and issue function requests to have its function code deployed and executed. For some commonly used and complex functionality (e.g., object identification in images or natural language parsing for audio streams), developers may choose to invoke functions from libraries offered by the service provider. An indication of the suitability of the FaaS paradigm for IoT workloads is that Amazon encourages developers to use serverless functions to develop Amazon Alexa Skills, e.g., customized programs to be invoked from smart home devices [13].

In the serverless paradigm, providers provision containers (or other sandboxed runtime environments) to serve as container instances. To execute a request, the appropriate container needs to be available on a server in the provider infrastructure. If a request arrives, and its associated container is not available yet, the container needs to be created, which implies in loading its image into memory, and initialized. If there is not enough memory available for the new container, the system needs to evict one or more of the currently hosted containers to free resources.

As stated by Baldini et al. [14], the system may 'scale to zero' some container instances to yield resources to respond to requests for other functions. Hence, a provider can accept more request types — i.e., offer a more extensive catalog of containers to customers — than it is able to host simultaneously given its capacity constraints. This capability may enable the providers to take more customers (and their functions), but it introduces a container management challenge: the decisions involved in the resource allocation, creation, and destruction of containers should be made such as the Quality-of-Service (QoS) customer specifications (e.g., requirements expressed as a target average or 99% request execution latency) are met. Resource management is still a challenge for traditional cloud computing providers, but for edge computing the ability to add physical servers to the edge network is much more constrained, if existent at all, and the provider is likely to operate under more stringent resource availability.

The dynamic instantiation of containers introduces a *coldstart overhead*: a container needs to be instantiated before it can serve a function request [2]. Though this overhead cannot be avoided for the first execution of each request type, evictions reintroduce the coldstart overhead for subsequent requests. While keeping containers in memory can significantly reduce the coldstart overhead, the entailed high cost in memory usage would lead to a trade-off involving resource contention, system performance, and service availability.

Our work addresses workloads from IoT devices. Many IoT applications deploy devices that capture data from the environment and periodically pass the data to a server for further processing, such as Smart Home systems [15], animal body signal sensors in ecological surveillance systems [16], temperature and humidity data for environmental control [17], and sensors on soil, tractors, and plant samples in smart agriculture deployments [18]. Other applications require periodic analysis of device data using application-specific methods, such as camera and movement sensor data in event/crowd management systems [19]. In such applications, devices issue requests in well-established periodic patterns.

The existing container management approaches target generic workload patterns, ignoring even highly predictable request patterns. In this work, we investigate the impact of exploiting the predictability opportunity that comes with periodic patterns. We evaluate the performance behavior of *de facto* container management methods such as OpenWhisk and Kubernetes under workloads that exhibit periodicity. We propose a new way to dispatch requests that takes into consideration the predictability introduced by periodic patterns.

2. BACKGROUND

2.1 Containers

A container image is a lightweight, stand-alone, executable package of software that includes code, runtime, system tools, system libraries, and settings [20]. Containers are runtime environments that provide isolation between applications. For example, different applications may have conflicting execution requirements; the isolation provided by containers can allow software to run with the desired library packages and even deploy customized versions. In contrast to the traditional hypervisor-based virtualization, container-based virtualization shares the operating systems (OS) kernel aross containers, and hence it has shorter instantiation time and smaller memory footprint.

The concept of OS-level virtualization is not new in operating systems. Similar ideas have been implemented in other systems, such as jails in FreeBSD [21], and zones in Solaris system [22]. In 2008, the LXC (LinuX Containers) [23] was released in Linux kernel. The two essential technologies for Linux containers – control groups, and namespaces – were available to general Linux users. Control groups allow a system to control and limit the resource usage for a process or a group of processes. Namespaces enable the system to grant privileges to different users or user groups.

Docker [24] is one of the container-based virtualization engines based on the LXC technology. Docker provides an ecosystem for users to leverage the advantages of container-based virtualization more easily. Using Docker, a user first creates a Docker image consisting of binaries, runtime environment, and library packages. The image is layer-based, and the host machine needs not to store the common layers of different images, therefore saving disk space. The user will instantiate the Docker image from disk to memory to run the application. Multiple instances of the application can be instantiated to scale out services. The system administrator can pause or stop containers to yield resources. When a container is paused, the container instance does not use CPU, but it still resides in memory. If a container is stopped, then the container instance may be removed from the memory. The system administrator will need to re-instantiate a stopped container to run its applications.

Container-based virtualization has been widely adopted in production. The promising lightweight virtualization makes it suitable for adapting to the dynamics of workloads. For example, containers have been used to carry out dynamic resource scheduling for MapReduce jobs in enterprise clusters to meet service level objectives [25], to create runtime environments for legacy codes [26], to achieve on-demand computational resource provisioning for mobile computation offloading [27], and just-in-time instantiation of services [28].

In 2007, Soltesz *et al.* [20] described the techniques used by Linux-VServer, a container-based virtualization tool. The discussion covered the implementations of resource sharing and security implementation of process and network. They made a performance comparison with Xen (hypervisor-based virtualization), and the results showed that the Linux-VServer can be more efficient and provide significant performance improvement for web-hosting workloads.

For the performance comparison between hypervisor-based and container-based virtualization, Xavier *et al.* [29] performed experiments to evaluate various compute virtualization technologies and concluded that LXC virtualization has a near-native performance on CPU, memory, disk, and network measurements.

One potential drawback of the container-based virtualization is the lack of isolation from the core operating system. Because containers share the same kernel, the isolation level of containers is not as complete as the hypervisor-based virtualization. Current container engines may need to improve [30] their file system, network and memory management in order to obtain performance and security guarantees.

In 2017, Manco *et al.* [31] proposed a design of lightweight VMs by using unikernels for specialized applications to achieve both isolation and efficiency. They analyzed the performance bottleneck of Xen servers and redesigned the Xen's control plane from centralized operation to a distributed one to reduce the interactions with the kernels. They presented LightVM and showed that its performance is faster than container-based virtualization. However, LightVM is optimized for an application. A container, on the other hand, is an application-agnostic runtime environment.

The container orchestration is another important problem in modern large cluster management of data centers, such as the Quincy project [32] and the Apollo project [33]. In 2015, Google shared their container orchestration platform, called Borg [34]. Borg aims at managing container services for complex production workloads, i.e., handling vraying resource requirements, service priorities, service lifecycle management, job failures, running across tens of thousands of machines, etc. The Borg project evolved to Kubernetes, an open-source project widely used in many production clusters.

2.2 Serverless computing

To address the fast evolution of application requirements, it has been proposed that applications be built by the the composition of simple and flexible functions. A Function-as-a-Service (FaaS) provides an environment for the deployment of such functions. The serverless paradigm allows developers to focus on building the functions, while the service providers take care of machines deployment and management. Current industrial leaders such as Amazon, Microsoft, and Google provide products for customers to deploy their functions on the cloud. Many open-source projects are also available: IBM OpenWhisk [6], OpenLambda [7], OpenFasS [8], and Kubeless [9].

Developers can write functions using various programming languages and customized libraries. Upon service requests, the service providers will create the runtime environment (generally with containers) and provision user functions. Customers only pay for the actual resources used while executing the function. The serverless paradigm is a good fit for current IoT applications because of its pricing model, the flexibility offered by its function-based software development, and the simplicity of scaling up services by deploying additional function instances.

As functions are provisioned within containers, the service providers are responsible for container creation, destruction, and even load balancing requests among machines. To better utilize its resources, service providers will stop containers to release resources for other functions.

Stopping a container to incurs a new challenge, the so-called coldstart overhead. The coldstart overhead is the instantiation time for a new container, which involves the time to create a new control group isolation, loading the container image from disk to memory, and application initialization. The overhead is introduced the first time the request arrives, but the container might be evicted due to resource limitation after a while; hence another coldstart overhead may be re-induced. Minimizing the probability of incurring a coldstart overhead while serving requests may impact positively the overall performance of service requests.

Lloyd *et al.* [35] and Wang *et al.* [2], in 2017 and 2018, conducted a series of experiments on AWS Lambda, and report a series of performance metrics. They find that as while the workload stress increased, AWS will provision more virtual machines to host container services. However, we consider this resource elastic is not existed in edge computing scenario.

There are many studies that improve the serverless computing performance. Pocket [36] and Locus [37] aim at providing cheap and efficient ephemeral storage to mitigate the possible excessive cost due to large amount of intermediate files generated by the computations. Mohan *et al.* [38] found that setting up network connections is a significant part of the container coldstart process. They proposed a method to reduce coldstart containers by pre-warming containers that connect to network.Akkus *et al.* [39] observed that AWS Lambda executes each function in individual containers, a design choice of higher isolation but with extra overhead and lower container utilization rate. They proposed *SAND*, a framework to analyze series of functions from the same users and to assign the placements of functions to the same containers whenever possible.

In addition to Docker containers, other runtime techniques are proposed for serverless computing, such as Google gVisor [40], LightVMs [31], WebAssembly [41], and serverless-optimized containers [42].

2.3 Edge computing

Envisioning the enormous growth of Internet-of-Things (IoT) applications that closely monitor and actuate on environments, processing data on the edge of the network becomes more and more critical due to constraints in network bandwidth, network latency, privacy control and so on [43]. For example, the Cisco Global Cloud Index [44] predicts that data produced by those "things" will reach 500 ZB, and 45% of IoT-created data will be processed at the edge of the network to reduce the burden of network bandwidth consumption. Another example is the real-time image processing with small devices, e.g., Google Glass, for using in wearable cognitive assistance applications. Ha *et al.* [45] built a prototype system that processes image data from Google Glasses in edge servers and measured both network bandwidth and latency reduction.

Shi *et al.* [43] define edge computing as the enabling technology allowing computation to be performed at the edge of the network. The "edge" can be any computing or network resources along the path between data sources and cloud data centers. Comparing to servers in a cloud data center, resource is often stringent at the edge and introduce more resource management challenges to system designers.

3. PROBLEM DESCRIPTION

3.1 Coldstart overhead

As discussed in Section 2.2, coldstart overhead is a well-known issue in serverless computing [7][14][2][35]. The instantiation of a container instance involves creating a new control group isolation unit in the hosting operating system, loading the container image from disk to memory, importing packages, and initializing the application [42]. Wang *et al.* [2] investigated the factors impacting performance for the major serverless computing offerings (AWS Lambda, Google Functions, and Microsoft Azure Functions). Their experiments demonstrated that the language used in the function implementation affects the coldstart latency. For example, the median coldstart latency of Python 2.7 functions is 167-171 ms while Java functions have higher latencies (824-974 ms) on AWS Lambda. Like other service, the request latency is a major consideration while developing a service.

3.2 State-of-the-art in open-source container management

Our analysis of the literature and open-source frameworks – OpenWhisk, OpenLambda, and Kubernetes – indicates that there are three types of container management policies to choose and a container for eviction instances: time-based, recency-based and size-based policies.

Time-based expiry policies, as its name implies, will retire container instances using a expiration time. Wang *et al.* [2] observed that AWS Lambda adopts the time-based expiry policy with elastic resource provisioning, i.e., AWS Lambda will provision new virtual machines as needed to serve more container instances and evict containers based on idle time ($>$ 27 minutes) to avoid wasting resources. They also observed the same behavior in Google Cloud Functions and Microsoft Azure Functions. Although the time-based policy is applicable on the cloud, the constraint is obvious: the servers (or the clusters) need to provide sufficient resources to instantiate all requested container instances elastically; otherwise, some requests may suffer from unacceptable latency due to the shortage. In edge computing environments, such resource elasticity is often unfeasible.

Recency-based policies order items by the time-of-access to catch the temporal locality of workloads. A typical representative is the least-recently used (LRU) policy. IBM OpenWhisk and OpenLambda both use LRU as their eviction policy. Using recency-based policy can free resources on-demand to create container instances for newly received requests.

Size-based policy is used by Kubernetes. From the Kubernetes document [46], when a resource type (e.g., memory) is "starved," the system will first evict containers that exceed the soft limit and then by container priority value, and finally by the "consumption" of the starved compute resource. In other words, if the same soft limit and priority applied toward all containers, the "size" of the containers will be the decisive factor for eviction candidate choosing in Kubernetes system.

Our literature search did not identify efforts in managing containers in edge servers. Based on documentation of the AWS Greengrass – the AWS Lambda solution on edge devices – its management of container instances appears to be quite primitive: the container instances either run indefinitely (possibly wasting resources) or on-demand (incurring high overheads). When running out of a resource (e.g., memory), the server cannot accept additional lambda function requests.

An edge server may need to offer more container instances because of privacy concerns, i.e., user-specific container instance. For example, Bai *et al.* [47] proposed edge-hosted personal services, in which each user will need a dedicated container instance (e.g., Docker container), to secure the privacy-sensitive data. Accordingly, the number of container instances soars as the number of users increases, making the system performance more sensitive to the container management method.

3.3 Terminology

We adopt the following terminology:

- **Function request**: <*uid, fn, timestamp*>. Requests invoked by IoT devices, where fn is the corresponding Function type and *uid* denote the user id (used in user-specific container instance).

- **Function type**: <*fn, coldstart time, memory size*>. The serverless services available. A container instance for a function type would take the *memory size* and *coldstart time* to instantiate.

3.4 Container management problem

When a service request is invoked to an edge server, the serverless system receive the request and queue it in the request pool, where stores the requests that need to be served. Note, there may be multiple requests received at the same time tick.

The server will first serve the service requests that already have the corresponding container instances provisioned, as there is no extra cost of provisioning. Then, the server looks into the request pool and decide which set of function type requested by service request will be served next, and provision container instances of the requested function types. During the provisioning, container instances that are not used for serving will be the victim candidate for the eviction.

In this work, we are focusing on a single edge server that offers serverless service toward nearby IoT devices. With the setup, we consider there is a container management problem in serverless computing, the problem choosing which containers to hold in memory at which point in time to minimize request latency and maximize overall system throughput.

The problem is similar to the classic caching problem [48]. For memory and storage systems, caches maximize the chance that a memory page or storage block is readily available when needed by the application. In content delivery networks, the system aims at maintaining the most popular web pages available to clients [49]. The cost of waiting for a page/block/web page to be retrieved from disk or through the network can have a significant impact on performance. Systems deploy methods designed to have available in a cache the resources needed when they are needed. The problem is also similar to the changeover cost problem [50] in operational research. Similarly, the management of containers offering Function-as-a-Service needs to maximize the chance that the appropriate container is ready for usage in memory when a request arrives.

Another motivation of this work is that currently adopted methods are generic solutions and are not designed for IoT workloads exhibiting periodicity patterns as we discussed in Chapter 1. We propose a new method that focus on exploiting the periodicity that may be present at IoT workload to mitigate the coldstart overhead.

4. METHOD

As described in the Section 3.4, our problem is that at any given time, a server needs to decide which function instances to provision from several function types, with considerations of the coldstart time and memory sizes of containers, and the periodicities of requests.

In this chapter, we will first describe our observations of the IoT workloads, and motivated by the observations, we propose a heuristic method.

4.1 Request serving order

We can make the following observations from several public IoT data sets (human physical activities monitoring [15], animal welfare monitoring [16], Chicago beach water quality monitoring [17], and smart farm control [18]):

- Many devices invoke requests in well-established periodic patterns
- Many devices invoke requests on punctual time manner, i.e., 00:00:00, 00:00:05, 00:00:10

We use a toy example to illustrate the motivations of our method and demonstrate the impact of our heuristic based on the *request serving order*. In Fig. 4.1, there are 4 types of function requests invoked at t_0. Periodicities of function 1 and 2 are 1-second, while periodicities of functions 3 and 4 are 2-second. Assume the system memory is able to hold only two container instances at the same time. The system needs to decide the *request serving order* and create function instances accordingly.

In this simple scenario, the system can serve the requests in two orderings. Order 1, depicted in Fig. 4.2, serves function requests 1 and 2 first (assuming the system can run two containers at the same time); after requests 1 and 2 are done, the system evicts function instances of 1 and 2, and provisions function instances of 3 and 4. The order 1 incurs two extra coldstarts because requests 1 and 2 are coming again at 1s.

Order 2, depicted in Fig. 4.3, serves requests 3 and 4 first, then serves requests 1 and 2. Because function instances for requests 1 and 2 are still in memory, serving additional requests 1 and 2 does not incur extra coldstarts.

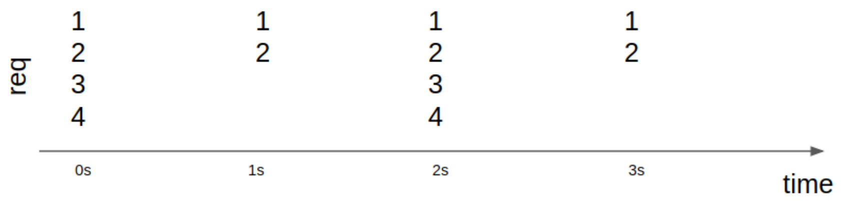

Figure 4.1: Toy example of requests. There are 4 request types invoked at timestamp 0s. Periodicity of requests 1 and 2 are 1-second, while periodicity of requests 3 and 4 are 2-seconds.

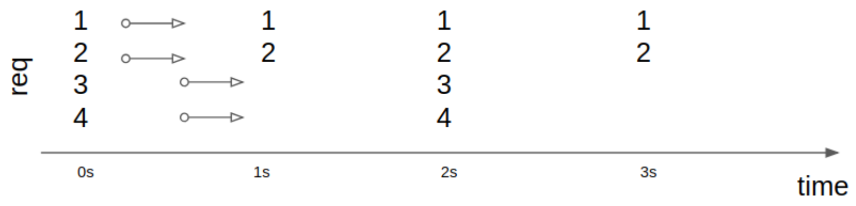

Figure 4.2: Order 1: serves request types 1 and 2 first (assuming the system can only run two containers at the same time); after requests 1 and 2 are done, the system evicts function instances of 1 and 2 and provisions function instances of 3 and 4. The order 1 incurs two extra coldstarts because requests 1 and 2 are coming again at 1s.

With the above toy example illustration, we also find current state-of-the-art solutions that focus on the eviction methods will not effectively mitigate coldstart overhead with the targeted IoT workload because the intervals (i.e., 1-second) is sufficient to serve all requests. Instead, the *request serving order* is an important factor in coldstart prevention of the targeted IoT workload. We proposed a new method focusing on deciding the request serving to exploit this observation.

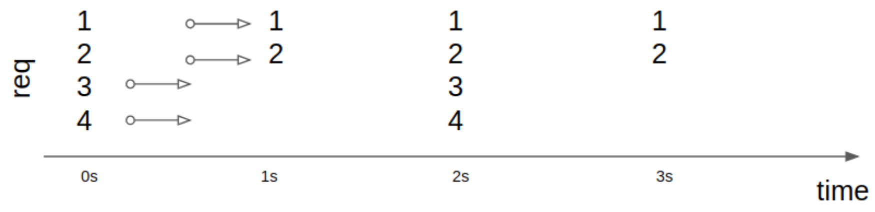

Figure 4.3: Order 2: serves requests 3 and 4 first, then serves function 1 and 2. Because function instances of function 1 and 2 are still in memory, serving another requests of function 1 and 2 does not incur extra coldstart.

4.2 Our method

Our method is a composite of three steps. We first analyze a simpler sub-problem by only considering two requests. Second, we demonstrate how to use the sub-problem to solve the original N-function serving order problem with O(N^2) complexity. Lastly, by using the IoT workload characteristics, we can further reduce the complexity into O($K \cdot S \cdot T$), where K is the number of distinct values for *number of queued function requests*, and S and T are two constants. S is the distinct values for *coldstart time* (in our experiment, S=3), and T is the number of the *time tick* we are considering.

We first create a sub-problem that only considers two requests (req_a and req_b) of two request types (a and b) and their next future requests (i.e., 4 requests in total). The goal of the sub-problem is to decide whether to serve a or b first, i.e., which serving order results in less total waiting time. With 4 requests and 2 request types, there are six combinations of request serving orders. We denote them as: *abab, baba, aabb, bbaa, abba, baab*.

We illustrate the request serving orders *aabb* and *abab* and their resulting waiting times next.

Let req_a and req_b be two request of two request types a and b. t_a and t_b are the arrival times of their next future requests. The system can easily get these times by determining the request periodicities. Finally, s_a and s_b are the coldstart times for the containers associated with a and b.

Fig. 4.4 shows the total waiting time breakdown of serving order *aabb*. The system has both req_a and req_b waiting for function instances at t_0, and by knowing the periodicities of the two requests, the system can predict that the arrival time of two requests will be t_a and t_b, respectively.

For brevity, we will use a ramp function notation $()_+$ in the formula:

$$(x)_+ = \begin{cases} 0 & \text{if x} < 0 \\ \text{x} & \text{if x} \geq 0 \end{cases}$$

Thus, the waiting time will be: (1) provisioning function instance, adding waiting time s_a to req_a; (2) waiting for the arrival of the new req_a, adding waiting time $(s_a - t_a)_+$ to the second req_a. If $t_a < s_a$, then the waiting time is 0; (3) evict function instance *a* and provision function instance *b*, adding waiting time max$(s_a, t_a) + s_b$ to req_b; in other words, req_b waits until two req_a are served and waits a coldstart time s_b to be served; (4) waiting for the arrival of the new req_b, adding waiting time $(max(s_a, t_a) + s_b - t_b)_+$ to req_b; Total waiting time is the sum of four terms: s_a + $((s_a - t_a)_+)$ + (max(s_a, t_a)+ s_b) + $(max(s_a, t_a) + s_b - t_b)_+$

We also compute the total waiting time of the serving order *abab* in Fig. 4.5. Total waiting time break down: (1) provisioning function instance, adding waiting time s_a to req_a; (2) evicting function instance *a* and provision function instance *b*, adding waiting time $s_a + s_b$ to req_b; (3) evicting function instance *b* and provision function instance *a*, adding waiting time $(2s_a + s_b - t_a)_+$ to req_a; (4) evicting function instance *a* and provision function instance *b*, waiting for the arrival of the new req_b, adding waiting time $(max(2s_a + s_b, t_a) + s_b - t_b)_+$ to req_b. Total waiting time is the sum of four terms: s_a + $(s_a + s_b)$ + $(2s_a + s_b - t_a)_+$ + $(max(2s_a + s_b, t_a) + s_b - t_b)_+$

We can determine the waiting time of other request serving orders by performing similar analyses. By listing the total waiting time of the six request serving order combinations, we can choose the order that results in the least waiting time.

A downside of this enumeration-based method is that it will often favor requests whose container has less coldstart time or favor requests whose periodicity is shorter. To mitigate this problem, we extend the previous enumeration by also considering the number of queued request, i.e., k requests asking the same type of function. We list all six request serving orders and their corresponding total waiting times in Table 4.1. By multiplying by k (the queue size) values, large

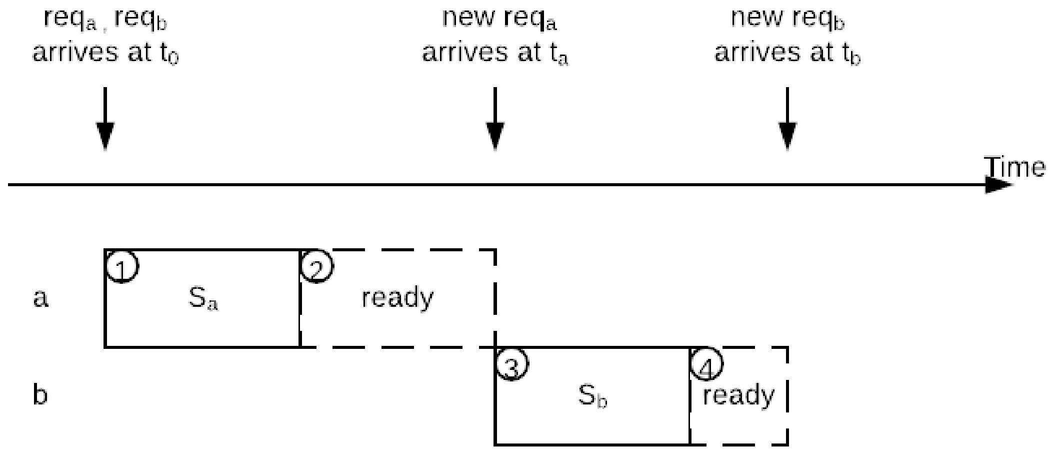

Figure 4.4: Serving order *aabb*. Total waiting time break down: (1) provisioning function instance, adding waiting time s_a to req_a; (2) waiting for the arrival of the new req_a, adding waiting time $(s_a - t_a)_+$ to the second req_a. If $t_a < s_a$, then the waiting time is 0; (3) evict function instance *a* and provision function instance *b*, adding waiting time max$(s_a, t_a) + s_b$ to req_b; in other words, req_b waits until two req_a are served and waits a coldstart time s_b to be served; (4) waiting for the arrival of the new req_b, adding waiting time $(max(s_a, t_a) + s_b - t_b)_+$ to req_b;

coldstart time containers and less frequent requests will eventually be picked up and served. For example, if we only consider serving orders *abab* and *baba* and k_b is large, then the system would favor the *baba* order because the $k_b \cdot (s_a + s_b)$ term (in *abab*) creates significantly more waiting time than $k_b \cdot s_b$ (in *baba*) does.

After computing the total waiting time of the six serving orders, the system selects the order that creates the minimum total waiting time as described in Algo. 1. The min_wait_order function will compute the total waiting time of the request serving order using the given orders, and return the minimum value of them. The output of the Algo. 1 is the decision of whether req_a or req_b should be served first.

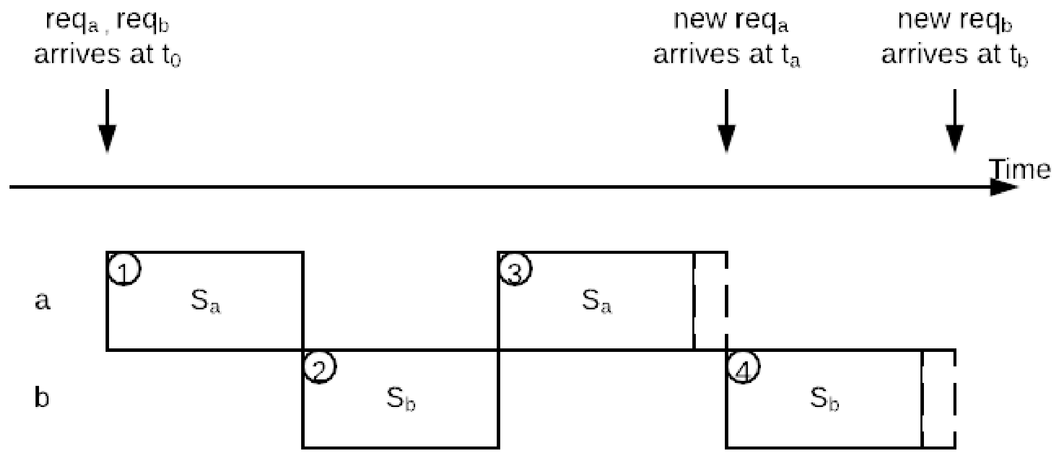
req$_a$, req$_b$ arrives at t$_0$
new req$_a$ arrives at t$_a$
new req$_b$ arrives at t$_b$
Time
a
b
1
S$_a$
2
S$_b$
3
S$_a$
4
S$_b$

Figure 4.5: Serving order *abab*. Total waiting time break down: (1) provisioning function instance, adding waiting time s_a to req_a; (2) evict function instance *a* and provision function instance *b*, adding waiting time $s_a + s_b$ to req_b; (3) evict function instance *b* and provision function instance *a*, adding waiting time $(2s_a + s_b - t_a)_+$ to req_a; (4) evict function instance *a* and provision function instance *b*, waiting for the arrival of the new req_b, adding waiting time $(max(2s_a+s_b, t_a)+s_b-t_b)_+$ to req_b; Total waiting time: s_a + $(s_a + s_b)$ + $(2s_a + s_b - t_a)_+$ + $(max(2s_a + s_b, t_a) + s_b - t_b)_+$

Algorithm 1 Determine serving order of two function types

Input: req_a, req_b; two function requests to two function types

Input: t_a, t_b; the next arrival time prediction using the periodicities

Input: s_a, s_b; coldstart time of function a and b

Input: k_a, k_b; number of queued request

Output: function type a or b

```
1: procedure COMPARE($req_a, req_b, t_a, t_b, s_a, s_b, k_a, k_b$)
2:     $time_a \leftarrow min\_wait\_order("aabb", "abab", "abba")$
3:     $time_b \leftarrow min\_wait\_order("bbaa", "baba", "baab")$
4:     if $time_a \leq time_b$ then
5:         return a
6:     else
7:         return b
```

Serving order	Expected waiting time formula
abab	$k_a \cdot s_a + k_b \cdot (s_a + s_b) + (2s_a + s_b - t_a)_+ + (max(2s_a + s_b, t_a) + s_b - t_b)_+$
baba	$k_b \cdot s_b + k_a \cdot (s_a + s_b) + (s_a + 2s_b - t_b)_+ + (max(s_a + 2s_b, t_b) + s_a - t_a)_+$
aabb	$k_a \cdot s_a + (s_a - t_a)_+ + k_b \cdot (max(s_a, t_a) + s_b) + (max(s_a, t_a) + s_b - t_b)_+$
bbaa	$k_b \cdot s_b + (s_b - t_b)_+ + k_a \cdot (max(s_b, t_b) + s_a) + (max(s_b, t_b) + s_a - t_a)_+$
abba	$k_a \cdot s_a + k_b \cdot (s_a + s_b) + (s_a + s_b - t_b)_+ + (max(s_a + s_b, t_b) + s_a - t_a)_+$
baab	$k_b \cdot s_b + k_a \cdot (s_a + s_b) + (s_a + s_b - t_a)_+ + (max(s_a + s_b, t_a) + s_b - t_b)_+$

Table 4.1: Waiting time of different request serving order. The serving order *abab* means: first provisioning function $instance\ a$ with coldstart s_a and serving req_a; and then evicting function $instance\ a$, creating function $instance\ b$ with coldstart s_b and serving req_b; evicting function $instance\ b$ to serve req_a; and finally evict function $instance\ a$ to serve req_b.
k_i, s_i, t_i are the ***number of queued request***, ***coldstart time of container*** i, and ***the next arrival time of the request type*** i.

To solve the original problem with N-functions, we use the Algo. 1 as the comparison function to compare N function requests against each other. Each winner of the comparison (e.g., req_a or req_b) will increment its score. Finally, with the scores, the system starts provisioning containers whose score is the highest until the available memory is insufficient to accommodate another new function instance. Then for the unselected function types (i.e., the containers not chosen for provision), we queue their requests, so their numbers of queued request (k) increase. It worth to mention that the higher k value is, in Algo. 1, the requests of the function type is more likely to win the comparison, so the system will more incline to pick the unselected function types in next run. We call it Look-Ahead Request Serving method (LARS) in Algo. 2.

Algorithm 2 Look-Ahead Request Serving order algorithm (LARS)

Input: REQ; N service requests

Input: S; coldstart time

Output: request serving order

```
 1: procedure SERVING ORDER($REQ, S$)
 2:     for each $req_i \in REQ$ do
 3:         for each $req_j \in REQ, \forall j \neq i$ do
 4:             $r \leftarrow$ COMPARE($req_i, req_j, t_i, t_j, s_i, s_j, k_i, k_j$)
 5:             score[r]++
 6:     score.sort()                                  ▷ sort from high to low
 7:     order $\leftarrow \emptyset$
 8:     while True do
 9:         $fn \leftarrow score.pop\_head()$
10:         if size of function instance $fn \leq$ free memory then
11:             $order = order \cup fn$
12:         else
13:             break
14:     return order
```

4.3 Reducing complexity by leveraging workload characteristics

The LARS method needs to compare N function types against each other, resulting in O(N^2) complexity. The complexity can be greatly reduced by using the two characteristics particular to IoT workloads.

First, the number of coldstart time values is limited. The study by Wang *et al.* [2] concluded that coldstart times are highly associated with runtime languages, e.g., the coldstart time for a node.js runtime is 150ms, while for a JAVA runtime it is 900ms. The second characteristic is that the arrival time of the requests appear in punctual time ticks, e.g., 00:00:00, 00:00:10, 00:00:20, etc. We can also use buckets and set a maximum look-ahead time to further restrict the number of time ticks in the computation.

Given these characteristics, most of the COMPARE function – i.e., Algo.1 – can reuse previous results. For example, req_a, req_b are function requests asking for a python runtime and a node.js runtime and their periodicities are 1-second and 2-seconds, respectively. Using the COMPARE function, the system determines that req_a should be served first. Then when system process another set of requests, req_c, req_d that are also asking for a python runtime and a node.js runtime with the same periodicities, the system can reuse the previous result of req_a, req_b and determine that req_c should be served first.

The complexity of performing at the COMPARE function in Algo. 2 can be reduced from O(N^2) into O($K \cdot S \cdot T$), where *S* and *T* are two constants. *S* is the distinct values for *coldstart time* (in our experiment, S=3), and *T* is a number of the *time tick* we are considering, leaving *K* (distinct values for *number of queued request*) is the only unbounded parameter. For most function types, the number of queued request (k) is 1, making the distinct value of *K* small enough. We observed that more than 50% of the computation can be saved in our experiment, while yielding the same results. The lower complexity makes our LARS method computationally more feasible.

5. EVALUATION

In this chapter, we evaluate our LARS method against other existing methods. First, we will use a handcrafted workload to inspect the factors of performance improvement. Then we evaluate the performance using two use case scenarios. Finally, we measure the magnitude of the computation cost saving described in Section 4.3.

5.1 Testbed and Configurations

We developed a simulation tool to emulate the serverless platform to test different configurations of function types and periodicities of function requests.

To evaluate the performance of our LARS method, three other methods were examined for performance comparison purpose.

First was the OpenWhisk method, which uses the least-recently used (LRU) policy to determine eviction candidates. We also used the OpenWhisk method as the baseline when calculating the performance difference between methods. The second was the method from Kubernetes [34], a container orchestration platform that is used by other open-source serverless projects to manage container creation and eviction. The eviction policy in Kubernetes is a size-aware policy, which will evict containers based on their resource usage. Specifically, in our work, we implemented the Kubernetes' policy by evicting the highest memory usage. We used as the third method a policy based on the Belady's algorithm for cache eviction [51], because the periodicity of requests provides future information of the "next arrival time of requests." Applying Belady's algorithm will guarantee the eviction candidate selection is optimal, but we out work concluded that exploring alternative request serving ordering has more impact. We called the Belady's based method the Eviction Oracle method.

As stated in Section 3.2, currently employed methods in serverless computing only focus on the eviction policy. We checked the source code of OpenWhisk and Kubernetes and found their serving order methods are both adopting the First-Come-First-Serve (FCFS) policy. Of course, those two systems were originally designed to run in the cloud, where limitation of the computational resources is vanishing. The request serving order is not their focus. Therefore, we implemented the FCFS method when simulating all comparison methods.

To highlight the performance improvement, we presented results by normalizing the waiting time against the OpenWhisk method (baseline method). Let T_{OW} and T_x be the average waiting time using the OpenWhisk (OW) method and the x method for the same workload. We defined an **improvement rate** with the equation:

$$rate = \frac{T_{OW} - T_x}{T_{OW}}$$

Each data point in the figures was generated by averaging 10 experiment runs, and for each run we generated 100,000 requests. We reported the average request waiting time for container instances.

5.2 Handcrafted workload experiment

Before diving into more realistic scenarios, we again used the handcrafted toy example as depicted in Fig. 4.1 to get an initial idea of how our LARS method performed on different workloads exhibiting periodicity.

The experiment setup was as follows. Given N IoT devices invoking periodic requests, the periodicities of each function requests were randomly assigned to either 1-second or 2-seconds. For container instances, we fixed the container memory to the same size and the coldstart time of containers to 0.5-seconds to simplify the environment.

We explored two variables: (1) **ratio**, the ratio of devices that send requests with 1 second and 2 seconds periodicities, which expresses the opportunities for varying the order in which requests are served and (2) **number of functions / capacity**, the total number of functions needed over the system memory capacity, which captures the stress to the system. We assumed that N devices will need N container instances, i.e., no container instance sharing.

We increased the number of devices (N) until the *eviction oracle* method showed no improvement. We pictured the results in Fig. 5.1 and Fig. 5.2. We didn't include the performance of the Kubernetes method because it chooses random victims due to all containers being equally sized.

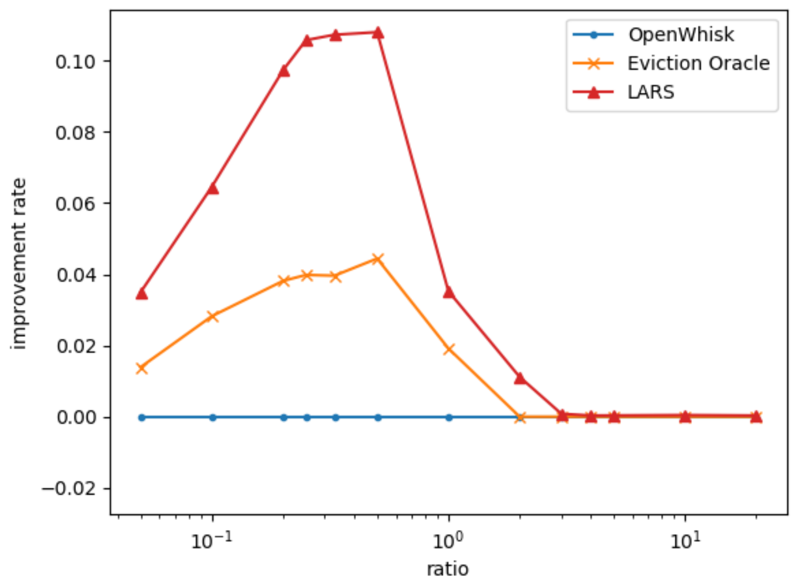

Figure 5.1: Ratio versus improvement rate. The x-axis is in log-scale. Note *ratio* > *1* means there were more devices sending requests using 1-second periodicity, while *ratio* < *1* means more devices sending requests using 2-seconds periodicity.

In the experiment, a higher ratio meant there were more devices sending requests using 1-second periodicity, while a lower ratio meant more devices were using 2-second periodicity. The right-hand side curves in Fig. 5.1 validate an intuition that if most requests are with high frequency, then the chance of the improvement is low. In contrast, the left-hand side curves reveal that opportunities exist. The improvement rate reached its highest point, about 10%, at *ratio* = *1/2*. The LARS method outperformed the eviction oracle method by 6%.

Fig. 5.2 also validates another intuition: the more number of functions is needed, the harder the problem is. Beginning with 1x of the system capacity, where the system can afford running all necessary container instances in memory, toward 1.6x capacity, the LARS method kept having higher

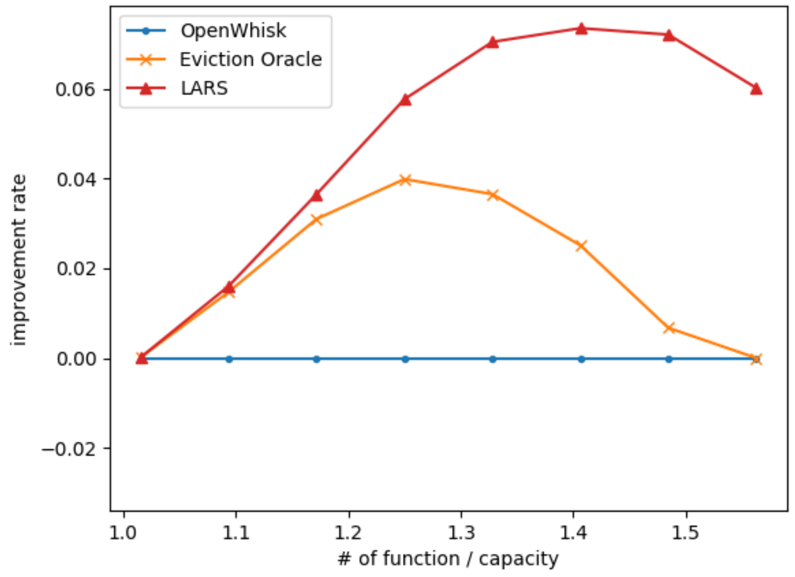

Figure 5.2: Number of functions over capacity versus improvement rate.

improvement rates. Even under such high stress, the LARS method still can find opportunities to make better container management decision.

5.3 Gym scenario

Now we investigated how the LARS method performs in emulated real-world use cases. We crafted two use case scenarios. The first created scenario was a smart gym.

The gym users are wearing personal devices to monitor bio-metric values during physical training. To protect such privacy-sensitive data, the serverless framework must provision dedicated container instances to serve requests for different users, i.e., more users generate more need for container instances. In this experiment, there were N users who all wear 4 personal devices with different periodicities (detail are listed in Table 5.1).

We also assumed 100 exercise machines that interact with users. The serverless framework provisions 100 container instances for these 100 machines as the users' interactions with the machines are also privacy-sensitive data.

For each supported serverless function types, we setup the container coldstart times by assigning to different runtime languages, referring the result in Wang et al. [2] study. Following were the coldstart times in our experiments for different runtime languages: Nodejs: 150ms, Python: 250ms, Java: 900ms.

The sizes of containers were arbitrary assigned. In essence, a vanilla Python container needs 16 MB memory space. For more sophisticated devices (e.g., exercise machines), we chose a larger container size.

The number of users range was from 100 to 1000. The corresponding ratio of the number of function over the system capacity was from 1.2x to 8.6x.

The improvement rate of each method as we vary the number of users is in Fig. 5.3. Note the improvement rate is related to the performance of the OpenWhisk method (baseline) for the same workload, which caused fluctuations to the curves of other methods.

The Eviction Oracle could only perform well in low number of users range, and soon lost its advantage. The LARS method had a good improvement rate at the first point, declined a little, and then bounced up. One possibility is that both Eviction Oracle and LARS, knowing the predictability information, were able to exploit the information in the low number of users. While the number of user was growing, the eviction strategy was insufficient to improvement the waiting time. The Kubernetes method cannot exploit the predictability and fails to make appropriate decisions.

In the end part (higher stress), eviction choices of the three other methods became useless because all remaining containers will be evicted to serve overwhelming requests for different function types. By deciding to alter the request serving order, LARS can intentionally provision containers that are likely to be used multiple times. The maximum improvement rate of LARS was 37% at the number of user was 600.

5.4 Smart Building Scenario

We created the second scenario by emulating a building (office building or apartment building) of multiple rooms inside. Each room is individually owned, so the serverless framework has to provision different container instances to different rooms even when they are using the same function type. The specification is listed in Table 5.2.

Device	Size of Container	Periodicity	Number of function needed
Heart rate sensor	16 MB	0.1 sec.	N
Pose monitor	32	0.5	N
Dashboard	64	1	N
Thermometer	16	5	N
Exercise machine	128	10	100 (fixed)

Table 5.1: Device setup for gym scenario

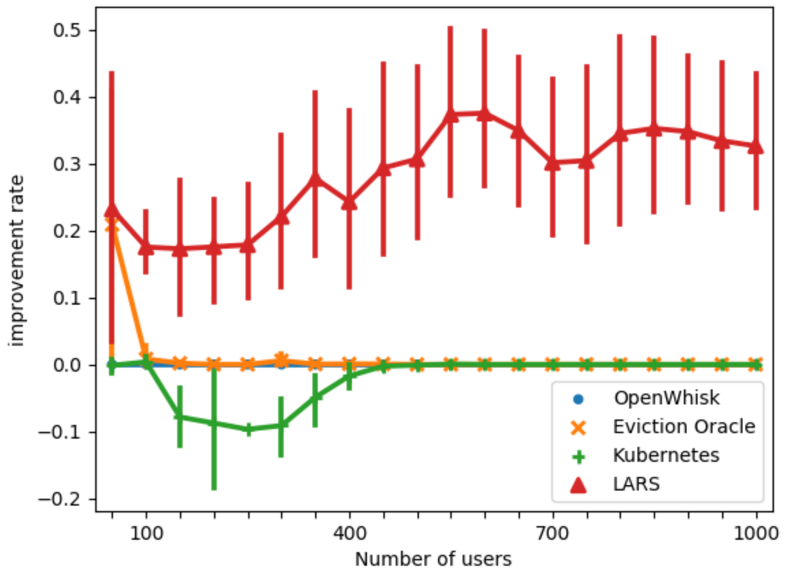

Figure 5.3: Improvement rate versus number of users in the gym scenario

The size of containers that provide function types for camera devices was 256 MB. We chose this memory size by inspecting a face recognition program, OpenFace [52]. We downloaded its container image of OpenFace from Docker Hub [53], and we found its memory consumption was about 250 MB. For the other functions providing services to two sensors, we simply assigned them small container sizes.

Unlike the gym scenario, in a room, there were multiple devices of the same type (e.g., camera) and the same type of device may have multiple periodicities. For example, a camera in the front door may invoke requests every 1 second, while a camera in the bedroom may just invoke every

Device	Size of container	Numbers per room	Periodicity	Number of function needed
Camera	256 MB	20	1, 5 sec.	2N
Thermometers	16	10	10, 20	2N
Humidity Sensors	16	10	30	N

Table 5.2: Device setup for Building scenario

5 seconds as the information is less time-sensitive. In our experiment, we randomly assigned the periodicity to the devices.

The building scenario simulated a more complex scenario where devices can share the same container instance as long as they are in the same room (i.e. same uid), but at same time, each device may issue function requests using different periodicities. We tested with the number of rooms from 100 to 500, and showed the result in Fig. 5.4. The corresponding ratio of the number of function over the system capacity varied from 1.6x to 7.8x.

The improvement rate was higher than that of the gym scenario. A major reason was that the IoT devices in the building scenario can share container instances. Therefore, the aggregated waiting time saving resulted in a better improvement rate.

5.5 Reduce the complexity by leveraging workload characteristics

Lastly, we were interested in how much algorithmic complexity LARS can save by using workload characteristics. Recall in Algo. 2, LARS computes the sub-problem of two requests of all N requests using the COMPARE function in Algo. 1, which results in $O(N^2)$ complexity. We argued that the complexity can be reduced to $O(K \cdot S \cdot T)$ when the distinct values for the *coldstart* and number of the *time ticks* are limited.

We output the *reuse rate* of the COMPARE function in the gym scenario and achieved the expected reduction in number of calculations. As illustrated in Fig. 5.5, the reuse rate was more than 90% when the number of function was about 500 and still over 40% even the number of functions was close to 4000. Another important observation is that even *K* (number of distinct values for *number of queued request*) is unbounded, the high reuse rate reflects that about half of the computations are reusable, showing that leveraging the workload characteristics to reduce to

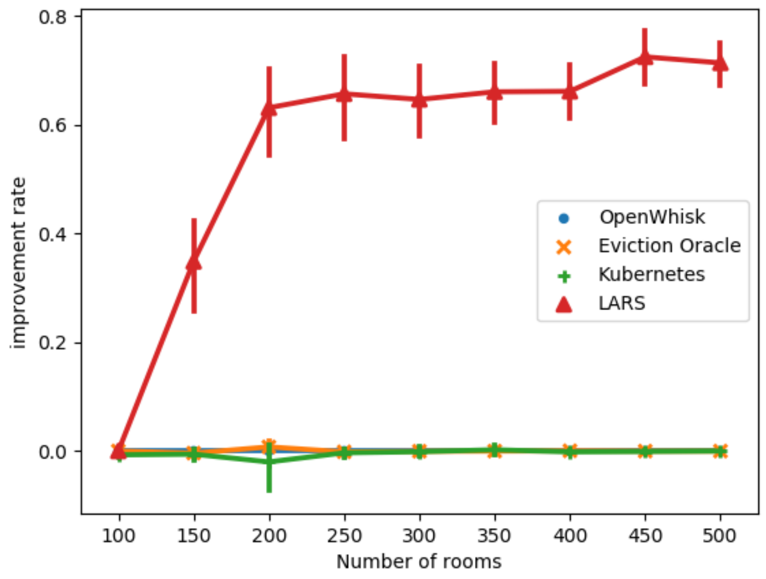

Figure 5.4: Improvement rate versus number of rooms in the building scenario

the computation cost of LARS is viable.

Note here we were reporting the reuse rate without a warm up base. If the function types and periodicities of requests are known in advance, a system administrator can pre-compute with the *K*, *S* and *T* to further reduce the computation overhead.

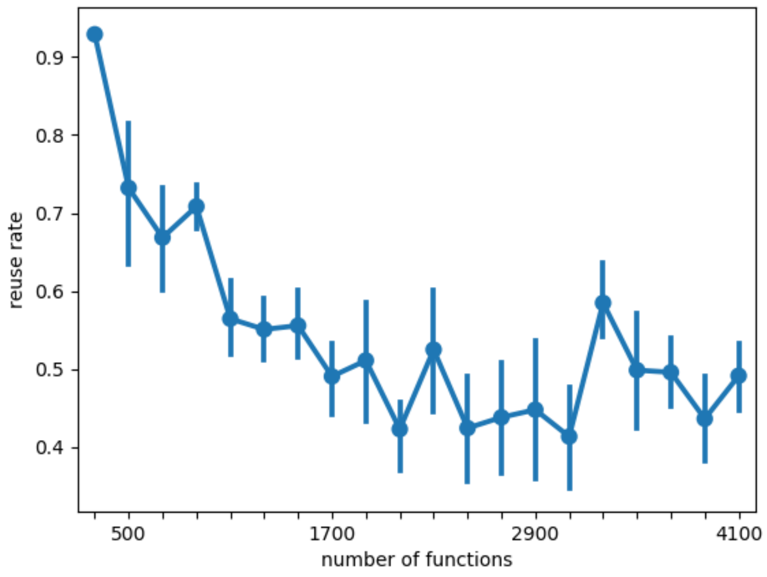

Figure 5.5: The compare function reused rate over different number of function

6. DISCUSSION AND CONCLUSION

6.1 Considering memory size in LARS

In the current design, the LARS method does not consider the memory size. We can extend LARS by using the 0-1 Knapsack algorithm to consider the memory size of containers as following.

Each function request will perform the original algorithm in Algo. 2 to get scores of each function type, so each function type has a tuple of weight and value, <*memory size, score*>. When determining the order, i.e. line 8 to line 13, we perform the 0-1 Knapsack algorithm. The objective is to maximize the score with the given memory capacity (i.e., max weight).

The complexity of performing the 0-1 Knapsack is $O(M \cdot N)$, where M is the memory capacity and N is the number of function types.

6.2 Considering more function types and requests in the sub-problem

The sub-problem only considers two function types and their one lookahead function requests. The sub-problem can also consider more function types with more lookahead requests.

However, we need to carefully evaluate the increased computation cost. Current LARS method needs to evaluate 6 serving orders. For two function types with two lookahead function requests, the number of all possible serving order is 20; for three function types with one lookahead function requests, the number of all possible serving order is 90; if we eagerly consider three function types with two lookahead function requests, the number of all possible serving order grows to 1680. We need to evaluate if the performance improvement worth the extra computation burden.

6.3 Caching decision-making for the collections of function requests

Although each device may invoke requests using different periodicities, the request in macro-level will form patterns, too. In other words, the same set of service requests (i.e. REQ in Algo 2) in the request pool will occur periodically. The system can, therefore, cache the request serving order using REQ as key, so the whole computation result can be reused, and hence greatly increase the computation burden of the LARS method.

6.4 Replace average waiting time by other objectives

In our algorithm, we want to minimize the average waiting time for function instances, by enumerating all possible combinations of the request serving orders. The same approach can apply with other objectives. For example, if the service level objective (SLO) is the objective, when performing the sub-problem computation, we can also pass the elapsed waiting time of requests, and increment the score of the request serving order that will satisfy the SLO requirement.

6.5 Conclusion

In this work, we explored the container management problem for serverless edge computing offerings. We investigated the currently employed methods and designed a new method that can successfully exploit the highly predictable patterns from the workloads. Our evaluations showed improvement in reducing the waiting time for the function instances.

REFERENCES

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[2] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 133–146, 2018.

[3] "Aws lambda." https://aws.amazon.com/lambda/. Visited in February 2019.

[4] "Google cloud functions." https://cloud.google.com/functions/. Visited in February 2019.

[5] "Microsoft azure functions." https://azure.microsoft.com/en-us/services/functions/. Visited in February 2019.

[6] "Ibm openwhisk." https://openwhisk.apache.org/. Visited in February 2019.

[7] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *HotCloud 16*, vol. 60, p. 80, 2016.

[8] "Openfaas." https://www.openfass.com/. Visited in October 2019.

[9] "Kubeless, the kubernetes native serverless framework." https://kubeless.io. Visited in October 2019.

[10] D. Oh, "7 open source platforms to get started with serverless computing." https://opensource.com/article/18/11/open-source-serverless-platforms. Visited in February 2019.

[11] "Amazon greengrass." https://aws.amazon.com/greengrass/. Visited in February 2019.

[12] “Azure iot edge.” https://azure.microsoft.com/en-us/services/iot-edge/. Visited in February 2019.

[13] B. Hussain, “Best practices for scaling your alexa skill using amazon web services.” https://developer.amazon.com/blogs/alexa/post/546ab5a1-1d1a-49c2-85a5-92ada3e6e907/best-practices-for-scaling-your-alexa-skill-using-amazon-web-services.

[14] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, “Serverless computing: Current trends and open problems,” in *Research Advances in Cloud Computing*, pp. 1–20, Springer, 2017.

[15] A. Reiss and D. Stricker, “Creating and benchmarking a new dataset for physical activity monitoring,” in *Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA ’12, (New York, NY, USA), pp. 40:1–40:8, ACM, 2012.

[16] M. Caria, J. Schudrowitz, A. Jukan, and N. Kemper, “Smart farm computing systems for animal welfare monitoring,” in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2017 40th International Convention on*, pp. 152–157, IEEE, 2017.

[17] Chicago-Park-District, “Beach water quality - automated sensors.” https://data.cityofchicago.org/Parks-Recreation/Beach-Water-Quality-Automated-Sensors/qmqz-2xku. Visited in February 2019.

[18] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. N. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman, “Farmbeats: An iot platform for data-driven agriculture.,” in *NSDI*, pp. 515–529, 2017.

[19] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The Case for VM-Based Cloudlets in Mobile Computing,” *IEEE Pervasive Computing*, vol. 8, pp. 14–23, Oct. 2009.

[20] S. Soltesz, H. Potzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors,” in

Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, (New York, NY, USA), pp. 275–287, ACM, 2007.

[21] "Freebsd jails." https://www.freebsd.org/doc/handbook/jails.html. Visited in October 2019.

[22] "Oracle solaris zones." https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm. Visited in October 2019.

[23] "Linux containers." https://linuxcontainers.org/. Visited in October 2019.

[24] "Docker." https://www.docker.com/. Visited in October 2019.

[25] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated SLOs for enterprise clusters," in *OSDI*, OSDI, 2016-11.

[26] A. Slominski, V. Muthusamy, and R. Khalaf, "Building a multi-tenant cloud service from legacy code with docker containers," in *2015 IEEE International Conference on Cloud Engineering*, pp. 394–396, 2015-03.

[27] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, "Container-based cloud platform for mobile computation offloading," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 123–132, 2017-05.

[28] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie, "Jitsu: Just-in-time summoning of unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 559–573, USENIX Association, 2015.

[29] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 233–240, 2013-02.

[30] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs containerization to support PaaS,” in *2014 IEEE International Conference on Cloud Engineering*, pp. 610–614, 2014-03.

[31] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, “My VM is Lighter (and Safer) Than Your Container,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, (New York, NY, USA), pp. 218–233, ACM, 2017.

[32] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: Fair scheduling for distributed computing clusters,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP ’09, pp. 261–276, ACM, 2009.

[33] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, “Apollo: Scalable and coordinated scheduling for cloud-scale computing,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 285–300, USENIX Association, 2014.

[34] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale Cluster Management at Google with Borg,” in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys ’15, (New York, NY, USA), pp. 18:1–18:17, ACM, 2015.

[35] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless Computing: An Investigation of Factors Influencing Microservice Performance,” in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, 2018.

[36] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.

[37] Q. Pu, S. Venkataraman, and I. Stoica, “Shuffling, fast and slow: Scalable analytics on serverless infrastructure,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), pp. 193–206, USENIX Association, Feb. 2019.

[38] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, “Agile cold starts for scalable serverless,” in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, (Renton, WA), USENIX Association, July 2019.

[39] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, “Sand: Towards high-performance serverless computing,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 923–935, 2018.

[40] Google/gVisor, “Open-sourcing gvisor, a sandboxed container runtime.” https://github.com/google/gvisor. Visited in February 2019.

[41] A. Hall and U. Ramachandran, “An execution model for serverless functions at the edge,” in *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI ’19, (New York, NY, USA), pp. 225–236, ACM, 2019.

[42] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Sock: Rapid task provisioning with serverless-optimized containers,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 57–70, 2018.

[43] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge Computing: Vision and Challenges,” *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, Oct. 2016.

[44] Cisco, “Cisco global cloud index 2015–2020,” *CISCO white paper*, 2015.

[45] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, “Towards wearable cognitive assistance,” in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’14, (New York, NY, USA), pp. 68–81, ACM, 2014.

[46] “Kubernetes evicition policy.” https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/#eviction-policy. Visited in October 2019.

[47] Y. Bai, P. Hao, and Y. Zhang, “A case for web service bandwidth reduction on mobile devices with edge-hosted personal services,” *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pp. 657–665, 2018.

[48] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, p. 19, USENIX Association, 1994.

[49] S. Borst, V. Gupta, and A. Walid, "Distributed Caching Algorithms for Content Distribution Networks," in *2010 Proceedings IEEE INFOCOM*, pp. 1–9, Mar. 2010.

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

[51] L. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[52] B. Amos, B. Ludwiczuk, and M. Satyanarayanan, "Openface: A general-purpose face recognition library with mobile applications," tech. rep., CMU-CS-16-118, CMU School of Computer Science, 2016.

[53] "Openface, dockerhub." https://hub.docker.com/r/bamos/openface/. Visited in Octobor 2019.