Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices with Serverless Computing

无服务计算：构建和运行应用时无需管理服务器等基础设施。

*Zijun Li, *†Quan Chen, *Shuai Xue, ‡Tao Ma, ‡Yong Yang, ‡Zhuo Song, *†Minyi Guo
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†Shanghai Institute for Advanced Communication and Data Science, Shanghai Jiao Tong University, China
‡Alibaba Cloud, China
{lzjzx1122, xueshuai}@sjtu.edu.cn, {chen-quan, guo-my}@cs.sjtu.edu.cn,
{boyu.mt, zhiche.yy, songzhuo.sz}@alibaba-inc.com

***Abstract*—While microservices that have stringent Quality-of-Service constraints are deployed in the Clouds, the long-term rented infrastructures that host the microservices are under-utilized except peak hours due to the diurnal load pattern. It is resource efficient for Cloud vendors and cost efficient for service maintainers to deploy the microservices in the long-term infrastructure at high load and in the serverless computing platform at low load. However, prior work fails to take advantage of the opportunity, because the contention between microservices on the serverless platform seriously affects their response latencies.**

由于每日负载 没有充分利用

响应延迟

Our investigation shows that the load of a microservice, the shared resource contentions on the serverless platform, and its sensitivities to the contention together affect the response latency of the microservice on the platform. To this end, we propose Amoeba, a runtime system that dynamically switches the deployment of a microservice. Amoeba is comprised of a contention-aware deployment controller, a hybrid execution engine, and a multi-resource contention monitor. The deployment controller predicts the tail latency of a microservice based on its load and the contention on the serverless platform, and determines the appropriate deployment of the microservice. The hybrid execution engine enables the quick switch of the two deploy modes. The contention monitor periodically quantifies the contention on multiple types of shared resources. Experimental results show that Amoeba is able to significantly reduce up to 72.9% of CPU usage and up to 84.9% of memory usage compared with the traditional pure IaaS-based deployment, while ensuring the required latency target.

混合执行引擎

感知争用的部署控制器

多资源争用监视器

①根据负载和无服务平台的争用情况 尾部延迟

②两种部署模式间快速切换

③定期量化对多种类型的共享资源的争用。

***Index Terms*—Serverless computing, Microservices, QoS**

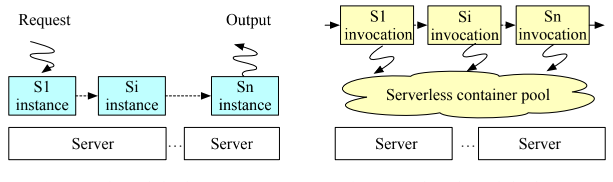
I. Introduction

Companies and individuals now start to build user-facing applications that have stringent Quality-of-Service (QoS) constraints with microservice architecture, instead of traditional monolithic architecture, for the high maintainability and testability [1]–[4]. Adopting microservice architecture, an application may be built with multiple loose-coupled fine-grained services from different developers that may run on different machines and communicate with each other through network.

细粒度

A fine-grained service can be deployed in either *IaaS-based* or *Serverless-based* modes in a Cloud. We refer a service that is light enough to run in the serverless computing platform as a *microservice* in this paper. As shown in Fig. 1, adopting IaaS-based deployment, the maintainers rent servers in the Cloud (Infrastructure-as-a-Service, IaaS), deploy and run the services

基础设施即服务。



(a) IaaS-based deployment (b) Serverless-based deployment

Fig. 1. Comparison of IaaS-based and serverless-based deployments.

in long term virtual machine instances. Adopting serverless-based deployment, the maintainers run the microservices (S_1, ..., S_n) in serverless manner (aka. Function-as-a-Service). Whenever a query is received, the corresponding function is invoked and executed with a container in the shared container pool. Adopting serverless-based deployment, maintainers pay for each function invocation instead of the whole infrastructure [18]. Cloud vendors already provide many serverless computing products, such as Amazon Lambda Function [1], Google Cloud Function [2], Microsoft Azure Functions [3], and Alibaba Function Compute [4].

However, the two deployment modes result in either low resource utilization or low peak load for emerging user-facing applications that often experience diurnal pattern [12], [13], [16] (the low load is less than 30% of the peak load). For a microservice, if IaaS-based deployment is adopted, its maintainers rent enough servers to support its peak load. In this scenario, the resources (e.g., cores and memory space) are wasted at a low load. On the contrary, if the serverless-based deployment is adopted, the service suffers from low peak load under the QoS constraint with the same amount of resources, due to the overheads of code/data loading, container cold start [28], and so on (to be explained in Section II). What is more, Some limits may increase as users' need growing, such as the concurrent request threshold, and restrict the max peak load in the serverless platform [11].

并发请求阈值，并限制无服务平台中的最大峰值负载。

Intuitively, from the perspective of service maintainers, it is cost-effective to adopt IaaS-based deployment at high load [27] and adopt serverless-based deployment at low load for emerging user-facing applications. Cloud vendors also benefit from this solution because of more tenants served with fewer computers. However, it is challenging to determine when to switch to IaaS-based deployment and vice versa.

Our investigation shows that the loads of a microservice

†Quan Chen and Minyi Guo are the corresponding authors.

另：SaaS：软件即服务。
PaaS 平台即服务。

at which the deployment should be switched from IaaS-based/serverless-based to serverless-based/IaaS-based are not identical. This is because the serverless computing platform runs function invocations from multiple applications, and containers in the container pool contend for shared resources (e.g., cores, IO, network). For instance, if there exists heavy IO contention in the serverless platform, an IO-bound microservice should not be switched to the serverless-based deployment even if its load is low. Meanwhile, a CPU-bound microservice can be safely switched to the serverless-based deployment.

Our insight is that *the load of a microservice*, the *real-time contention on the serverless platform*, and *the sensitivities of the microservice on multiple shared resources* together determine the appropriate time to switch the deployment mode. An offline methodology is not applicable to find the appropriate loads to switch the deploy mode, because the appropriate loads are affected by real-time contention behaviors.

离线的方法不适合.

Based on the above insight, we propose **Amoeba**, a runtime system that is comprised of a *hybrid execution engine*, a *multi-resource contention monitor* and a *contention-aware deployment controller* to minimize resource usage while ensuring the QoS of microservices. The execution engine enables the switch of IaaS-based deployment and serverless-based deployment at runtime with negligible overhead. The contention monitor periodically checks the contention status on shared resources in the serverless computing platform. Based on the real-time contention and carefully built performance models, the deployment controller determines when to switch the deployment modes. Amoeba applies for both microservices separately managed by maintainers and the traditional monolithic services if they are small enough to be expressed in the form of functions with the serverless computing platform.

The main contributions of this paper are as follows.

主要贡献

- **A real-system runtime mechanism that bridges IaaS-based and serverless-based microservice deployments.** It enables the automatic deploy mode switch for minimizing resource usage while guaranteeing the QoS.
- **A novel method that quantifies the contention on shared resources.** We design delicate functions that run on the serverless platform to quantify the contention on the shared resources in the serverless platform.
- **A precise contention model that predicts the QoS of user-facing microservices in the serverless platform.** For a microservice, the model considers its sensitivities to the contention, its load, and the contention on the shared resources when performing the prediction.

The experimental results show that Amoeba significantly reduces up to 72.9% of CPU usage and up to 84.9% of memory usage while satisfying the QoS requirement of user-facing microservices compared with the traditional IaaS-based deployment.

II. Investigating Microservice Deployments

In this section, we investigate the effectiveness of the IaaS-based and serverless-based deployments in satisfying the required QoS and minimizing resource usage of microservices.

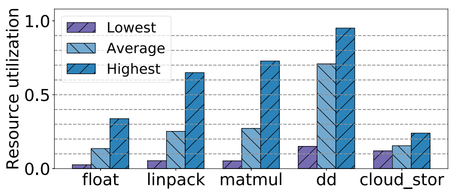


Fig. 2. The CPU utilization of the benchmarks with IaaS-based deployment.

A. Investigation Setup

We use five microservices in FunctionBench [19], a microbenchmark for serverless function service, as the microservice benchmarks to investigate the effectiveness of the traditional IaaS-based deployment and the new serverless-based deployment. The used benchmarks show various pressure and sensitivity on CPU, memory, IO and network resources, thus cover a large spectrum of real-system microservices. In the investigation, the load patterns of the benchmarks are generated according to the load trace from Didi [5] to emulate real-system load fluctuates patterns. The actual fluctuate pattern does not affect the analysis in this section. The experimental platform and the characteristics of the benchmarks will be shown in Section VII-A.

B. Effectiveness of IaaS-based deployment

Adopting IaaS-based deployment, each microservice is packed into a *virtual machine* (*VM*) image. Once the VM is started, it occupies the rented resources (cores and memory space) during its lifetime, no matter it is busy in processing user queries or not [26]. This mechanism enables short end-to-end latency of user queries because the service is always ready to process user queries, and the QoS of a microservice is always satisfied, because the maintainers rent enough resource for the peak load of the service.

In this scenario, we seek to answer the question of whether the IaaS-based deployment can effectively utilize the rented resources or not. To emulate the real system scenario that the service maintainers tend to minimize the cost, we deploy each benchmark on the infrastructure that is "just-enough" to guarantee the QoS of the benchmark under the peak load.

节约成本,

Fig. 2 shows the lowest, the average, and the highest CPU utilization of the benchmarks if IaaS-based deployment is adopted. The lowest, the average, and the highest CPU utilization of the benchmarks range from 2.6% to 15.1%, from 13.6% to 70.9%, and from 24.1% to 95.1%, with IaaS-based deployment. The low average CPU utilization turns into cost ineffectiveness in consequence.

Observed from Figure 2, the highest CPU utilization of some benchmarks at the peak load is still small (e.g., *float*, *cloud_stor*). This is mainly because these benchmarks have tight QoS targets or bottleneck in the network. Scheduling more queries to a core with low utilization may result in their QoS violation due to the context switch overhead and port contention between the queries. *Traditional IaaS-based deployment results in poor resource utilization for user-facing applications that experience diurnal load patterns.*

由于上下文切换开销和查询之间的端口争用 将大量的查询调度到低利用率的核心可能会导致QoS问题

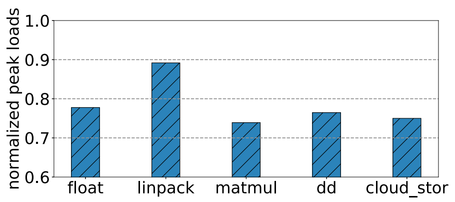


Fig. 3. The achievable peak loads of the benchmarks adopting serverless-based deployment compared with IaaS-based deployment with the same resources.

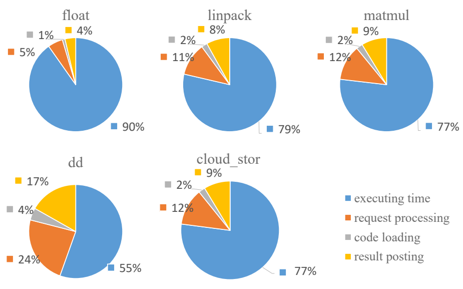


Fig. 4. The breakdown of the latencies of queries in the benchmarks with serverless-based deployment.

C. Effectiveness of serverless-based deployment

To eliminate the resource waste, serverless computing (aka. Function-as-a-Service) is proposed. Adopting the serverless-based deployment, maintainers provide executable code of a microservice that is able to be processed by temporary containers in the serverless computing platform ran by Cloud vendors. Once a user request is received, the platform launches/reuses an appropriate container to run the corresponding function to serve the request. Because multiple microservices at low load can be consolidated on the same serverless computing platform, the total resource usage and the cost is low.

However, serverless-based deployment supports lower peak loads of user-facing applications with the same amount of resources compared with IaaS-based deployment. Fig. 3 shows the peak loads of the benchmarks while ensuring their QoS with serverless-based deployment normalized to their counterparts with IaaS-based deployment. Observed from Fig. 3, the achieved peak loads of the benchmarks with serverless-based deployment range from 73.9% to 89.2%. The low peak loads originate from the extra overhead of user authentication, authorization, code loading, container cold start, etc. in the serverless-based deployment.

To explain this problem, Fig. 4 shows the latency breakdown of a user query of each benchmark if the query is executed on the serverless computing platform. Observed from this figure, the extra overheads (processing, code loading, and result posting) take 10% to 45% of a query's end-to-end latency, even if the queuing effect and the container cold start overhead is not counted in this experiment. The extra overhead results in the lower peak loads of user-facing applications with serverless-based deployment compared with the traditional IaaS-based deployment.

额外开销高的话不划算

It is not smart to always deploy a microservice in the serverless computing platform due to the high overhead. *For a user-facing application that experiences diurnal load pattern, it is better to adopt IaaS-based deployment at high load and adopt serverless-based deployment at low load.*

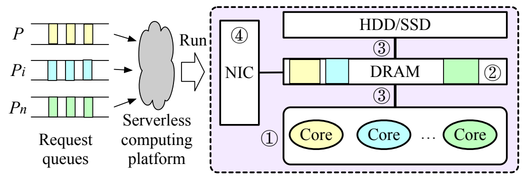


Fig. 5. Factors that determines the tail latency of a user-facing application P that adopts serverless-based deployment. NIC: Network Interface Card.

D. Factors that affect the QoS with serverless computing

However, it is nontrivial to identify the appropriate loads at which to switch the deploy modes, because multiple microservices run on the same serverless computing platform in public Clouds. The impact of queries parallelism and resource contention between the containers that execute functions from different microservices results in the unstable latencies of the microservices.

Fig. 5 shows an example scenario that a user-facing microservice P runs on a serverless computing platform. As shown in this figure, queries of multiple user-facing applications are submitted to and executed by the serverless computing platform. Observed from this figure, the end-to-end latencies of P's queries are affected by three factors besides their actual processing time.

- **The queuing time**. If the load of P is high, Queries queue up for the free containers in the serverless platform. The long queuing time results in P's long-tail latency.
- **The contention on shared resources**. If other applications co-located with P have a high load, they result in severe interference on the performance of P. Specifically, as shown in the right part of Fig. 5, the applications contend for ① **cores**, ② **memory space**, ③ **IO bandwidth** and ④ **network bandwidth**. The size of memory space limits the total number of containers that can run concurrently in the serverless platform.
- **P's sensitivities on the resource contention**. The performance degradation of P due to the shared resource contention is also affected by its sensitivities to the contention on the shared resources. For instance, the performance of P may not be seriously degraded if it is only sensitive to the core contention, but only heavy network contention exists on the serverless platform.

According to the above analysis, the latencies of P's queries on the serverless computing platform vary even if its load is stable, because loads of the co-located applications (e.g., P_i, P_n) may change quickly at runtime. **There is not a fixed load at which to switch P to serverless-based mode due to the varying contention situation.** Similarly, there is not a fixed load at which to switch P to IaaS-based mode.

In addition, switching P into serverless-based deploymen-t also affects the latencies of existing applications in the serverless computing platform. It is also crucial to ensure that switching P to serverless-based deployment would not result in the QoS violation of the current user-facing applications.

E. Challenges in minimizing resource usage and ensuring QoS

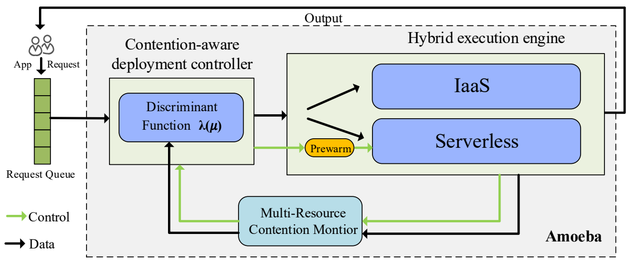
To minimize the resource usage while ensuring the QoS of user-facing applications, Amoeba should be able to identify the appropriate loads at which to switch the deploy mode based on runtime information. Amoeba faces three key challenges.

- **The QoS of a user-facing application on the serverless computing platform is not known before the application is actually deployed on the platform.** If we monitor the QoS of a user-facing application after the deployment is switched to serverless-based, it may already suffer from QoS violation during the monitoring. Worse, the inappropriate switch may also result in the QoS violation of the current applications in the serverless platform.
- **The complex contention behaviors on multiple resources.** While user-facing applications on the same serverless computing platform contend for multiple resources, a user-facing application's sensitivities to different resources are unknown. In addition, the performance degradation of a user query due to the contention is not the simple accumulation of its degradations due to the contention on each type of resource, because the contention on multiple resources is not independent.
- **The fluctuating of user-facing applications' loads.** It 波动 is possible that some user-facing applications have to be switched to IaaS-based deployment due to the burst load and vice versa. Amoeba should be able to capture the load change, and quickly switch the deployment with low overhead at runtime.

III. DESIGN OF AMOEBA

Fig. 6 shows the design overview of Amoeba, a runtime system that is comprised of a *contention-aware deployment controller*, a *hybrid execution engine* and a *multi-resource contention monitor*. Amoeba is a system designed for Cloud vendors. For a user-facing application, based on its load and the real-time shared resource contentions on the serverless computing platform, the deployment controller predicts the QoS of the application if it is switched to serverless-based deployment, and determines the deployment mode for the application. The execution engine enables fast switch of the two deployment modes, and routes query to either the IaaS-based service instances or the serverless platform accordingly. The contention monitor periodically quantifies the resource contention on various resources in the serverless platform by invoking the carefully designed functions. These functions are referred to as *contention meters*. The quantified contentions on different resources are sent to and used by the deployment controller to perform the QoS prediction.

Take a microservice S_a that experiences diurnal load pattern as an example. When S_a is first submitted to deploy on the Cloud, its executable function for the serverless platform, packed VM image for the IaaS, and required resource configuration for the IaaS-based deployment (enough to serve S_a's peak load) are provided. Amoeba minimizes the resource usage of S_a while ensuring its QoS in the following steps.



Output
App
Request
Request Queue
Contention-aware deployment controller
Discriminant Function $\lambda(\mu)$
Hybrid execution engine
IaaS
Serverless
Prewarm
Multi-Resource Contention Montior
Control
Data
Amoeba

Fig. 6. Design of Amoeba.

1) The contention-aware deployment controller adopts IaaS-based deployment mode for S_a to guarantee its QoS by default. Amoeba also routes queries of S_a to the serverless platform, and collects the CPU resource consumption of a query of S_a. Based on the load of S_a, the feedback from the multi-resource contention monitor, the deployment controller periodically determines whether to switch the deployment mode based on a precise QoS model. The challenging part here is precisely predicting the QoS of S_a before it is switched to the serverless platform that has complex contentions.

2) If the controller decides to switch the deployment of S_a to the serverless-based mode, the execution engine warms enough containers for the queries of S_a before the switch actually happens. After the containers are warmed, the new queries of S_a are routed to the serverless platform, and the IaaS platform releases the resources after all it's allocated queries completed. Similarly, the execution engine boots the VM instances for S_a before the deployment is switched to the IaaS-based mode at high load. The challenge here is how to effectively eliminate the cold start overhead that often results in the QoS violation of microservices with short QoS targets.

3) The contention monitor periodically launches contention meters to run on the serverless platform, and quantifies the contention on the shared resources (cores, memory, IO, network) of the platform. The contention monitor runs as a daemon process on the serverless platform. Based on the quantified contentions, the deployment controller of S_a predicts the QoS of S_a if it is switched to serverless-based deployment. It is challenging to quantify the performance of S_a under complex contention, because S_a shows different sensitivity to the contention on different resources.

It is worth noting that Amoeba does not need users to provide any extra information besides the executable function for the serverless computing platform, compared with the traditional pure IaaS-based deployment. In addition, if deploying S_a on a serverless platform show short enough latency for S_a but results in the QoS violation of any other microservice on the platform, the deployment of S_a will not be switched to the serverless-based mode.

IV. CONTENTION-AWARE DEPLOYMENT CONTROLLER

Due to the constantly changing workloads in the serverless platform, it is impracticable for Cloud vendors to manually switch the deployment modes. Especially, whether a microservice can be safely switched to the serverless-based deployment depends on the contention situation on the serverless platform.

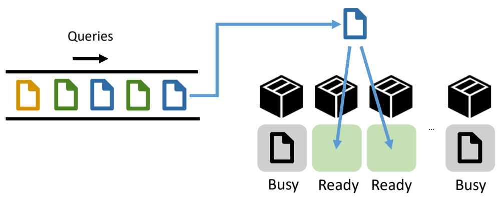


Fig. 7. Sketch of M/M/N queueing model, where queries queue for containers.

To solve the first challenge mentioned in section II-E, we first analyze the processing logic in the serverless computing platform. Fig. 7 shows the logic diagram of the queuing and processing. When a query Q_{blue} is submitted to the FIFO queue, it will get involved by a ready container. If no container satisfies, the serverless system will launch a new container and allocate resources and runtime to it, which is also called cold start. And then query Q_{blue} is invoked and executed and container status is turned to busy until invocation is done.

The above problem can be formalized to be a producer-consumer problem. Queuing theory, which has a wide range of applications, especially in communication systems, computers and storage systems serve as an appropriate approach to this problem. This theory demonstrates high adaptability in reduced cost and maximum benefit where queue exists. Moreover, the assumptions in the M/M/N model is consistent with the concept in serverless computing. So we pick this model as our basic discriminant function.

A. M/M/N Queueing Model as Discriminant Function

The M/M/N model applied in this paper makes the following assumptions: (1) one waiting queue and infinite query capacity in the system. (2) queries arriving interval obeys the exponential distribution of λ. (3) container processing time and query arrival time are both independent. Table IV-A shows the parameters and symbols used in the queueing model.

TABLE I
SYMBOLS USED IN THE M/M/N QUEUEING MODEL.

λ	The interval obeys negative exponential distribution.
n	The number of containers in the serverless platform.
μ	The processing capacity of each container.
π_k	The probability that there are k queries in the system.

When $\rho = \frac{\lambda}{n\mu} < 1$, $\pi_0 = [\sum_{k=0}^{n-1} \frac{(n\rho)^k}{k!} + \frac{(n\rho)^n}{n!(1-\rho)}]^{-1}$ and its stable distribution is shown in Equation 1.

$$\pi_k = \begin{cases} \frac{(np)^k \pi_0}{k!}, k = 1, 2, \ldots, n-1 \\ \frac{n^n \rho^k \pi_0}{n!}, k = n, n+1, \ldots \end{cases} \tag{1}$$

Let W represent the waiting time in the queue under the steady state ($\rho < 1$). Because $P\{W = 0\}$ equals to $p\{X < n\}$ if $W = 0$, so we can derive Equation 2 and Equation 3.

$$\begin{aligned} P\{W=0\} &= p\{X<n\} = 1-p\{X \geq n\} = 1 - \sum_{k=n}^{\infty} \pi_k \\ &= 1 - \sum_{k=n}^{\infty} \frac{n^n \rho^k}{n!} \pi_0 = 1 - \frac{\pi_n}{1-\rho} \end{aligned} \tag{2}$$

$$\begin{aligned} P\{0<W<t\} &= \sum_{k=n}^{\infty} P\{w<W<t|X=k\}P\{X=k\} \\ &= \sum_{k=n}^{\infty} \pi_k \int_0^t \frac{(n\mu)^{k-n+1} x^{k-n}}{\Gamma(k-n+1)} e^{-n\mu x} dx \\ &= \int_0^t \pi_n \sum_{k=n}^{\infty} \frac{(n\mu x\rho)^{k-n}}{(k-n)!} n\mu e^{-n\mu x} dx \\ &= \frac{\pi_n}{1-\rho} [1 - e^{-n\mu(1-\rho)t}] \end{aligned} \tag{3}$$

If $t > 0$, the distribution function of W is derived from the above equation and can be simplified to be Equation 4.

$$\begin{aligned} F_w(t) &= P\{W = 0\} + P\{0 < W < t\} \\ &= 1 - \frac{\pi_n}{1-\rho} e^{-n\mu(1-\rho)t}, t > 0 \end{aligned} \tag{4}$$

Equation 5 shows the discriminant function to determine whether the current deploy mode is able to satisfy the QoS of a microservice. In this equation, T_D represents QoS target and r represents r-ile latency of a miroservice.

$$\lambda(\mu) = n\mu + \frac{ln[\frac{(1-r)(1-\rho)}{\pi_n}]}{T_D - \frac{1}{\mu}} \tag{5}$$

If $\lambda \leq \lambda(\mu)$, the QoS of the microservice can be satisfied when it is switched to the serverless platform. Otherwise, the IaaS-based deploy mode should be kept for the microservice.

In order to restrict the resources consumption in the serverless platform [11], an upper limit for container quantity $n_{max} = min\{\frac{1}{\delta} \frac{M_0}{M_1}\}$ is introduced. It is limited by the resource consumption of a query mentioned in Section I.

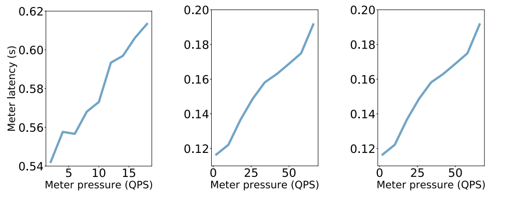
Except for the processing capacity of each container (denoted by μ), other parameters in Equation 5 can be given by the user or collected in the background. However, as mentioned in Section II-D, the contentions on multiple types of resources are not independent. Applications' contention for multiple resources brings about the variation of the value of μ. So in the next subsection, we will discuss in this respect.

B. Quantifying Shared Resource Contention

As mentioned before, a serverless platform may host multiple microservices simultaneously. In this scenario, the concurrent microservices contend for multiple types of shared resources, and the QoS of a microservice when it is deployed in the serverless platform cannot be predicted without μ.

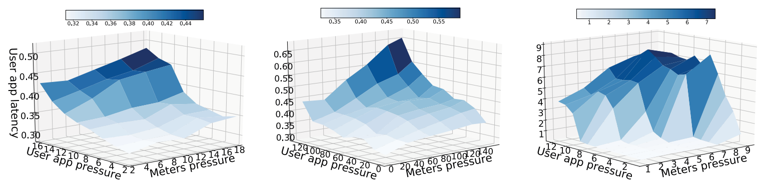
To predict the latency increase of a microservice query due to the contention on the shared resources (e.g., core, IO, and network), we design three delicate functions as *contention meters* to capture the pressure value on the shared *core*, *IO bandwidth*, and *network bandwidth* in the serverless platform. Specifically, based on the contention meters, we predict the QoS of a microservice by three steps.

1) Profiling: For each contention meter, we run it alone on the serverless platform, adjust the load of the meter, and record the latencies at different loads (Fig. 8).It characterizes the contentiousness of each microservice in terms of the pressure on the corresponding resource. When a contention meter is deployed on the serverless platform, the longer the latency of a contention meters query is, the higher the pressure microservices put on corresponding resource is.



(a) CPU meter (b) IO meter (c) Network meter

Fig. 8. The latency variation of CPU/IO/Network contention meters with their loads (meter pressure).



(a) Sensitivity to CPU (b) Sensitivity to IO (c) Sensitivity to net

Fig. 9. Latency surfaces of a microservice that measure its sensitivity to the CPU/IO/network contention.

As mentioned in section II-D, microservice's sensitivities on the resource contention is also a factor degrading performance. So in addition, for each microservice, we co-locate it with each of the contention meters on the serverless platform, adjust the loads of the microservice and the pressure of the contention meter, and built it three latency surfaces that shows how the performance of each microservice degrades as pressure increases in two dimensions. Fig. 9 shows the latency surfaces of an example microservice. For a long-running microservice, it is acceptable to profile it for the great resource usage reduction in future.

2) *Measurement:* Amoeba runs the contention meters in the background of the serverless platform with a low load, and records the latency of contention meters as a set $l = \{l_{CPU\_Memory}, l_{IO\_bandwidth}, l_{net\_bandwidth}\}$. By comparing the latency of a contention meter with the curve in Fig. 8, we can quantify the meter pressure on the corresponding resource from all microservices on the serverless platform (denoted by $P = \{P_{CPU\_Memory}, P_{IO\_bandwidth}, P_{net\_bandwidth}\}$).

Based on the pressure P, the load of a microservice V_u, and the latency surfaces $L(P, V_u)$ of the microservice as shown in Fig. 9, Amoeba is able to predict the latency affected by CPU, IO and network:L_{CPU}, L_{IO}, and L_{net} respectively. For easing of description, we term them the set $\{L_1, L_2, L_3\}$.

3) *Calculating:* Let α represents the overhead of executing queries, L_0 represents the solo-run latency of the microservice without interference from other applications, and μ_n represents the value of the processing capacity of each container. Equation 6 calculates the iterative value of μ_n.

$$\mu_n = \frac{1}{L_0 \sum_{i=1}^{3} w_n \frac{L_i}{L_0} + \alpha} \tag{6}$$

Before the switch happens, previous queries routed to the serverless platform serve to estimate the value of the weight w_0. So we introduce the multi-resource contention monitor. This module will turn the closely related variables into as few new variables, meanwhile, make them pairwise unrelated. Feedback from the monitor will update w_0 to $w_1 \ldots w_n$ along with the new calculated $\mu_1 \ldots \mu_n$ as Amoeba runs.

V. Hybrid Execution Engine

There are two key components in the hybrid execution engine of Amoeba: IaaS computing platform, and serverless computing platform. IaaS computing platform manages to start up a VM instance, allocate runtime, load code, configure API gateway, and shutdown during the life cycle. Serverless computing platform typically refers to function services.

A. Minimizing the Impact of Container Code Start

最小化容器代码的影响

One of the biggest complaints against serverless computing is the relatively long cold start time [6]. In the background, the serverless platform uses containers to encapsulate and execute the functions. When a function is invoked for the first time, 冷启动 the serverless platform needs to start up a new container, initialize it, and run the function. All these steps make up a cold start. Most serverless containers spend one to three seconds on a cold start, and it will result in the long latencies of the queries [28]. An efficient mechanism to reduce the cold start overhead is reusing the container. After a container completes the execution of a function, instead of shutting it down immediately, it is alive for a certain period of time. In OpenWhisk, they are known as warm containers that perform the best in terms of latency and throughput. 热容器

Amoeba spawns prewarmed containers to anticipate the user load, in which case queries initialize the warm containers to eliminate cold start. The hybrid execution engine continually monitors the control signal from the controller and informs the prewarm module to keep enough warm containers for later functions/queries. Let n, QoS_t, V_u represent the number of the prewarmed containers, the QoS target of a microservice and the speed/concurrency of user queries in the microservice respectively. Because of that most serverless platforms allow only one execution at a time in a container [10], so $\frac{n}{QoS_t}$ represent the minimal speed/concurrency of user queries that can be processed. So n should satisfy Equation 7.

$$\frac{n-1}{QoS_t} < V_u \leq \frac{n}{QoS_t} \tag{7}$$

One exist contradiction is that too many prewarmed containers result in expensive costs during the creation on the one hand, but fewer ones result in potential QoS violation on the other hand. The value of n calculated by Equation 7 ensures that the prewarmed containers is enough and leaves space for creating more containers for burst invocations. Though the prewarm mechanism may cause additional resource usage in the serverless platform, it still performs well in resource-saving. The result will be shown and discussed in Section VII.

B. Switching the Deployment Mode

切换部署模式

When the controller determines to switch the deployment modes, a control signal S_{pw_A} is first sent to the prewarm module of the engine. Once the signal is received, the execution engine starts to prepare the production environment, where containers are prewarmed for serverless and isolated

VM machine started up for IaaS. At the same time, the controller monitors acknowledgement passed from multi-resource contention monitor, which indicates space ready for routing queries to another computing platform.

In summary, the transformation only occurs after acknowledgement received by a controller, which means the server is launched and ready. When queries are routed to the former computing platform again after a while, the controller will pass the prewarm module a shutdown signal S_{sd_A} to release the resources occupied by the isolated VM machine or containers. As everything is sound, the gate opens for the next invocation.

VI. Multi-resource contention monitor

As observed in section IV-B, the capability to capture load change counts when performing the deployment switch quickly at runtime. In this section, we will introduce the multi-resource contention monitor and its mechanism in updating the weight value in the controller.

A. *Calibrating Performance Interference*

While a large number of microservices may co-run in a serverless computing platform, they may contend for multiple types of shared resources. More importantly, many resource variables can be correlated, which also makes it expensive to find the relationships between the tail latency of a microservice and the resource usage using traditionally neural networks. To better predict the performance interference of a microservice on the serverless platform, we adopt PCA (Principal Component Analysis), which serves as the most widely used data dimension reduction algorithm. 数据降维 PCA method merges close-related variables into as few new variables as 合并密切相关的变量 possible and makes them pairwise unrelated. 成对 不关联 In this way, it is possible to represent the interferences due to the contention on multiple types of resources with the interference due to fewer comprehensive contention indicators.

In our implementation, to apply PCA to update the weights in the deployment controller, the heartbeat package that contains latency and resource pressure data during the sample period is sent from the execution engine to contention monitor. The data in the package serves to generate the latency performance by three steps, which are previously described in section IV-B. Through PCA, we select the principal components that can cover the most variance of the data. At the same time, the weights will be updated from w_0 to $w_1 \dots w_n$ in Equation 6 and then forwarded to the deployment controller. Finally, the value of μ_n will converge to the real processing capacity of containers bounded with the user application.

In our experiment, three resource dimensions were involved. In a production environment, Cloud vendors may take more diverse resources contention into consideration. PCA will significantly reduce the cost of the training process in updating weights compared with that in the traditional neural network.

B. *Determining Sample Period*

As discussed in the previous subsection, in order to evaluate the performance of QoS, a heartbeat package containing all the query processing latency during the sample period is sent from the execution engine to the feedback module.

TABLE II
Hardware and software setup

	Configuration
Node	CPU: Intel Xeon Platinum 8163@2.50GHz Cores: 40, L3 shared cache: 32MB DRAM: 256GB, Disk: NVME SSD Network Interface Card (NIC): 25,000Mb/s
Network	25,000Mb/s Ethernet Switch
Software	IaaS-based deployment: VM+Nameko Serverless-based deployment: OpenWhisk Memory of serverless-based container: 256MB Operating system: Linux with kernel 3.10.0

However, even though we make the containers prewarmed before invocation, cold start by accident is inevitable. In this case, QoS violation on this account during the short period will induce the misjudge of the controller in spite of enough resources in serverless-based deployment to handle queries.

To avoid the misjudgment in the deployment controller, T needs to satisfy Equation 8, where $cold\_start, QoS_t, t_{exec}$ and e represent cold start time, QoS target, query execution time and the allowed error scope, respectively.

$$T > \frac{cold\_start - QoS_t + t_{exec}}{(1-e)QoS_t} \tag{8}$$

Depends on the allowed error e, the sample period T changes. If the allowed error is small, Amoeba has to sample the contention on the serverless platform more frequently.

VII. Evaluation of Amoeba

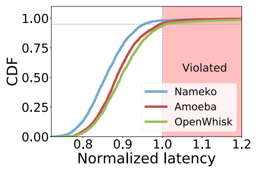
In this section, we first evaluate the performance of Amoeba in reducing resource usage while ensuring the QoS of microservices. Then, we show the effectiveness of the contention monitor and the hybrid execution engine. Lastly, we report the overhead caused by Amoeba.

A. *Experimental Setup*

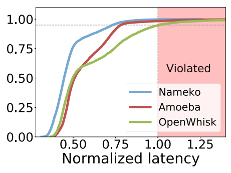
We evaluate Amoeba on a 3-node cluster where the nodes are connected with a 25Gb/s Ethernet switch. Table II shows the hardware and software configuration of each node.

In Amoeba, we implement the serverless-based part of the hybrid execution engine by modifying Apache OpenWhisk, an event-driven serverless computing platform [7], and implement the IaaS-based part using Nameko [8] that is hosted by virtual machines to emulate the in-production environment. For fairness of comparison, we compare Amoeba with Nameko [8], the pure IaaS-based deploy system, and OpenWhisk [7], the pure serverless-based deploy system to show the effectiveness of Amoeba in this section. In the 3-node experimental cluster, we use one node to be the IaaS node, one node to deploy the shared serverless platform, and one node to generate queries, run the deployment controller and the contention monitor.

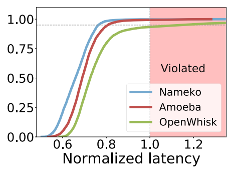
We use the benchmarks in FunctionBench [19] to evaluate Amoeba. As shown in Table III, the benchmarks stress on various resources, thus cover a large spectrum of real-system microservices. To add a slight pressure with the diurnal pattern on serverless, we select *float*, *dd*, and *cloud_stor* to run with a lower peak load as the background service by carefully designed parameters. Then we evaluate each benchmark with a diurnal pattern whose peak load is set high enough to arise transformation in an execution engine. For each benchmark



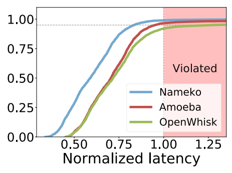
(a) float



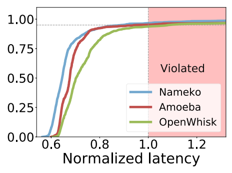
(b) linpack



(c) matmul



(d) dd



(e) cloud_stor

Fig. 10. The cumulative distribution of the benchmarks' latencies normalized to their QoS targets with Amoeba, Nameko, and OpenWhisk.

TABLE III
THE BENCHMARKS USED IN THE EXPERIMENTS.

Name	Sensitivity of loads			
	CPU	*Memory*	*Disk I/O*	*Network*
float	high	high	-	-
matmul	high	high	-	-
linpack	high	high	-	-
dd	medium	medium	high	-
cloud_stor	low	low	medium	high

and background service, its load pattern is configured based on the query trace from Didi [5] to simulate real-system scenarios.

In the following experiment, the resource usage of a benchmark is normalized to its resource usage with the long term IaaS-based deployment, and the QoS of a benchmark is defined to be the 95%-ile latency of the benchmark.

B. QoS and resource usage of Amoeba

In this experiment, we evaluate Amoeba in reducing resource usage while ensuring the QoS of microservices.

Fig. 10 shows the cumulative distribution of the benchmarks' latencies normalized to their QoS targets with Amoeba, Nameko, and OpenWhisk. Observed from this figure, the 95%-ile latencies of the benchmarks are shorter than the QoS targets with both Nameko and Amoeba. On the contrary, OpenWhisk results in the QoS violation of *matmul*, *dd* and *cloud_stor*.

OpenWhisk is not able to guarantee the QoS of the benchmarks at high load. When a load of a benchmark is high, the resource contention on the serverless computing platform may result in the long latency of its queries. On the contrary, Nameko is able to guarantee the QoS because the rented infrastructure is enough to host the peak loads of the benchmarks. Amoeba is also able to guarantee the QoS of the benchmarks because it adaptively switches the deployment mode. When the load of the benchmark is high, the processing of the queries is switched to the IaaS platform.

Observed from Fig. 10, for each benchmark, the curve of Amoeba is close to the curve of OpenWhisk for queries with short latencies and is close to the curve of Nameko for queries with long latencies. This is because Amoeba adopts serverless-based deployment for a benchmark at low load (the latency is short in this case) and adopts IaaS-based deployment at high load (the latency is relatively long in this case).

While both Amoeba and Nameko are able to guarantee the QoS, Fig. 11 shows the normalized resource usage of the benchmarks with Amoeba compared with Nameko. Observed from Fig. 11, Amoeba reduces the CPU usage of the benchmarks ranging from 29.1% to 72.9%, and reduces the memory usage of the benchmarks ranging from 30.2%

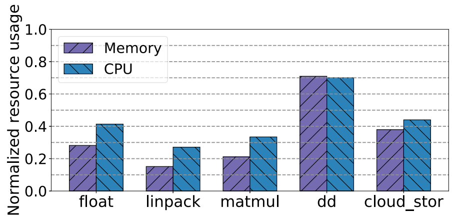
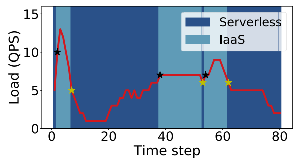
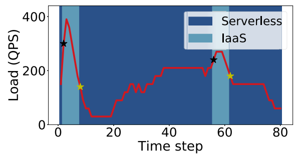


Fig. 11. The normalized resource usage of the benchmarks with Amoeba compared with Nameko.



(a) float



(b) dd

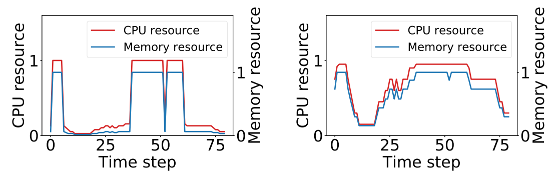
Fig. 12. Timeline of the deploy mode switch with Amoeba.

to 84.9%, compared with Nameko that adopts IaaS-based deployment. Amoeba is able to reduce the resource usage of the benchmarks because it uses a serverless computing platform to run the queries at low load and releases the IaaS resources after all its queries complete.

To better illustrate the way that Amoeba works, Fig. 12 shows the deploy mode switch timeline of two representative benchmarks *float* and *dd*. In the figure, the red line shows the load variation of the corresponding benchmark, the dark blue part means Amoeba adopts serverless-based deployment for the benchmark, the light blue part means Amoeba adopts IaaS-based deployment for the benchmark, the black/blue stars represent the load at which to switch to serverless-based/IaaS-based deployment. Observed from this figure, the loads at which to switch to the serverless-based deployment or the IaaS-based deployment are not identical. This is mainly because the QoS of a microservice on the serverless computing platform depends on not only the load of the microservice, but also the interference from other services on the same serverless computing platform. Amoeba is able to capture the contention on the serverless platform and adaptively switch the deploy mode of the benchmarks.

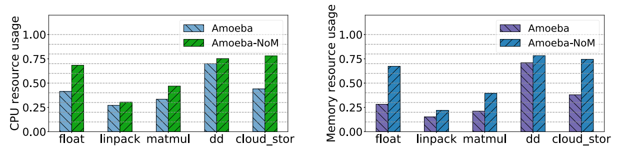
Besides the timeline of switching the deploy mode, Fig. 13 shows the resource usage timeline of two representative benchmarks *float* and *dd* with Amoeba. The usage change patterns of *matmul* and *cloud_stor* are similar to the pattern of *float*, and the pattern of *linpack* is similar to the pattern of *dd*. We omit them due to the limited space.

Observed from Fig. 13, the timeline shows two different change patterns. If a microservice has tight QoS requirement



(a) float (b) dd

Fig. 13. The timeline of resource usage variation with Amoeba.



(a) CPU usage. (b) memory usage.

Fig. 14. Resource usage of the benchmarks with Amoeba and Amoeba-NoM normalized to their resource usage with Nameko.

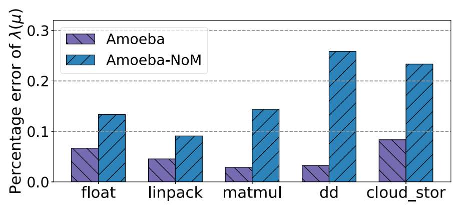


Fig. 15. Average error of the discrimination function $\lambda(\mu)$ of the benchmarks with Amoeba and Amoeba-NoM (smaller is better).

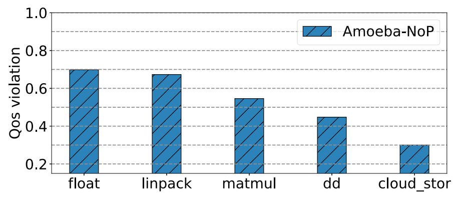


Fig. 16. QoS violation of the benchmarks with Amoeba-NoP.

(the processing time of a query is close to the QoS target), is sensitive to shared resource contention, or the code for each query is serial, the resource usage may change suddenly even if the load of the microservice changes smoothly (Fig. 13(a)). This is because the number of containers when performing the deploy mode switch (denoted by $n_{container}$) is far less than the maximum number of the supported containers in the serverless computing platform n_{max} ($n_{container} \ll n_{max}$). Otherwise, the resource usage changes smoothly with the load of the microservice (Fig. 13(b)). *Amoeba effectively reduces the resource usage of microservices with various characteristics while ensuring their QoS.*

C. Effectiveness of the contention monitor

To investigate the effectiveness of the contention monitor, we implement *Amoeba-NoM* that disables the PCA correction analysis in the contention monitor of Amoeba (Section VI-A).

Fig. 14 shows the resource usages of the benchmarks with Amoeba and Amoeba-NoM. Observed from this figure, Amoeba-NoM results in higher resource usage compared with Amoeba for all the benchmarks. It increases the CPU usage and memory usage by up to 1.77x and 2.38x, respectively. The poor performance of Amoeba-NoM originates from the late switch from the IaaS-based deploy mode to the serverless-based mode at low load. In order to guarantee the QoS of a microservice, Amoeba-NoM has to pessimistically assume that the QoS degradations of a query due to the contention on each of the shared resources are accumulated. In this case, Amoeba-NoM switches the deploy mode to serverless-based that shows lower resource usage later than Amoeba.

Benefit from the PCA correction that calibrates the weights of w_n and μ_n in the deployment controller, the existence of the monitor helps to improve the accuracy of discriminant function. $\lambda(\mu_n)_{max}$ can be calculated by Equation 5, which represents the theoretical switch point. And λ_{real} achieved by enumeration points out the real switch point. Comparing the calculating switch point $\lambda(\mu_n)$ with the real switch point λ_{real}, Fig. 15 presents errors with the requirement of 95%-ile latency of queries both in Amoeba and Amoeba-NoM. Amoeba reduces the max prediction errors from 25.8% to 8.3% and the min error from 9.1% to 2.8% on average.

Observed from Fig. 14, *linpack* and *dd* achieve similar CPU and memory resource usage. This is mainly because they are not sensitive to load change. For Cloud vendors, they can employ Amoeba-NoM to applications that are not sensitive to the load change. In this case, operators take less responsibility in managing clusters.

D. Effectiveness of the container prewarm strategy

In order to evaluate the effectiveness of the container prewarm module for the serverless platform in Amoeba, we implement *Amoeba-NoP*, a variation of Amoeba that does 不预热 not prewarm containers before switching to the serverless platform. In Amoeba-NoP, the queries are directly routed to the serverless platform accordingly.

Fig. 16 shows the QoS violation of the benchmarks with Amoeba-NoP. Observed from Fig. 16, 29.9% to 69.1% of the queries suffer from QoS violation in the benchmarks with Amoeba-NoP. This is because of the cold start overhead of containers usually beyond the QoS target of the benchmarks. *The container prewarm module in the execution engine can effectively eliminate the QoS violation due to the container cold start at the serverless platform.*

E. Overhead of Amoeba

Amoeba uses PCA correction and contention meters to monitor the sensitivity of a microservice on the contention and quantify the contention. However, although the training process can be arranged in other machines, the contention meters are co-located with microservices on the serverless platform to capture the resource features. This method brings in the overhead in the serverless platform, which may cause deviation during the switching process.

We trace the background pressure of the contention meters and analyze the impact. We define the overhead by CPU utilization on average during the sample period and set that each contention meter runs for 1 query per second (QPS). By our observation, for CPU-Memory sensitive, IO sensitive and network sensitive contention meter, they assume 1.1%, 0.5%

and 0.6% resource overhead respectively. Actually, these three meters can be scheduled in a round time trip, which means the total overhead caused by them is no more than 1.1%.

VIII. Related Work

There is a large amount of prior work on improving the resource utilization while ensuring the QoS of user-facing applications with diurnal patterns [14], [15], [21], [22], [30].

Some work proposed to co-located best-effort applications with user-facing applications at low load to improve resource utilization [14], [15], [21], [22], [29]. They either identify "safe" co-location pairs where the co-location does not result in QoS violation [14], [22], [29], or explicitly allocate shared resources between the co-located user-facing applications and best-effort applications [21], [31]. However, they mainly target private datacenters and rely on the users to provide the best-effort applications for improving resource utilization. They are not applicable to public Clouds.

Some other prior researches like Kubernetes [9], support capacity for Kubernetes-based elastic applications [25]. There are also researches on performance [10], [20] and improving serverless computing platform [17], [23]. To achieves better resource efficiency, lower latency, and more elasticity, a new serverless computing system is presented called SAND [10]. Lin and Glikson [20] implemented a pool of function instances to address the cold start problem in serverless architectures. However, these researches are based on resources shared platform, where container scheduling may have poor performance in peak load due to the contention or resources in our scenario.

Besides the above researches on a single platform, a tool is proposed to automate the process of generating serverless-microservices based high-level architecture design for an intended business application [24]. And our concentration on the resources optimization and Qos requirements are orthogonal to their motivation.

IX. Conclusions

Amoeba minimizes the resource usage by proper switching between IaaS-based and serverless-based deployment for microservices with diurnal load patterns. Designed as a runtime system, Amoeba enables the automatic deploy mode switch by a contention-aware deployment controller, a hybrid execution engine, and a multi-resource contention monitor. In terms of total resource usage of emerging user-facing workloads, Amoeba shows an obvious reduction in CPU and memory usage by up to 72.9% and 84.9%, respectively, compared with the traditional IaaS-based deployment. Meanwhile, it still can promise the 95%-ile latencies shorter than the QoS targets.

Acknowledgment

This work is partially sponsored by the National R&D Program of China (No. 2018YFB1004800), the National Natural Science Foundation of China (NSFC) (61632017, 61872240, 61832006, 61702328), and Alibaba.

References

[1] aws.amazon.com/cn/lambda/. Oct., 2019.
[2] cloud.google.com/functions/. Oct., 2019.
[3] azure.microsoft.com/en-us/services/functions/. Oct., 2019.
[4] www.alibabacloud.com/zh/products/function-compute. Oct., 2019.
[5] github.com/fortianyou/Di-tech. Oct., 2019.
[6] serverless.com/blog/2018-serverless-community-survey-huge-growth-usage. Oct., 2019.
[7] openwhisk.apache.org/. Oct., 2019.
[8] nameko.readthedocs.io/en/stable/. Oct., 2019.
[9] kubernetes.io/. Oct., 2019.
[10] I. E. Akkus, R. Chen, I. Rimac, M. Stein, and K. Satzke. SAND: Towards high-performance serverless computing. In *ATC*, pages 923–935, 2018.
[11] 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*, pages 1–20. Springer, 2017.
[12] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE micro*, (2):22–28, 2003.
[13] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.
[14] Q. Chen, Z. Wang, J. Leng, C. Li, W. Zheng, and M. Guo. Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters. In *ICS*, pages 272–283. ACM, 2019.
[15] Q. Chen, H. Yang, and M. Guo. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *ASPLOS*, pages 17–32, 2017.
[16] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
[17] S. Hendrickson, S. Sturdevant, T. Harter, and V. Venkataramani. Serverless computation with openlambda. In *HotCloud*, 2016.
[18] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: a berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.
[19] J. Kim and K. Lee. Functionbench: A suite of workloads for serverless cloud function service. In *CLOUD*, pages 502–504. IEEE, 2019.
[20] P.-M. Lin and A. Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221*, 2019.
[21] D. Lo, L. Cheng, R. Govindaraju, and P. Ranganathan. Heracles: Improving resource efficiency at scale. In *ISCA*, pages 450–462. ACM, 2015.
[22] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Micro*, pages 248–259. ACM, 2011.
[23] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *ICDCSW*, pages 405–410. IEEE, 2017.
[24] K. Perera and I. Perera. Thearchitect: A serverless-microservices based high-level architecture generation tool. In *International Conference on Computer and Information Science*, pages 204–210. IEEE, 2018.
[25] K. Takahashi, K. Aida, T. Tanjo, and J. Sun. A portable load balancer for kubernetes cluster. In *HPC Asia*, pages 222–231. ACM, 2018.
[26] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *the 10th Computing Colombian Conference*, pages 583–590. IEEE, 2015.
[27] M. Villamizar, O. Garces, L. Ochoa, H. Castro, Salamanca, et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *CCGrid*, pages 179–182. IEEE, 2016.
[28] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *ATC*, pages 133–146, 2018.
[29] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*, pages 607–618. ACM, 2013.
[30] H. Yang, Q. Chen, M. Riaz, and Z. Luan. Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp. In *ISCA*, pages 133–146, 2017.
[31] W. Zhao, Q. Chen, and H. Lin. Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus. In *IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 653–663, 2019.