HyperFaaS: A Truly Elastic Serverless Computing Framework

Jingyuan Zhang[*] Ao Wang[*] Min Li[†], Yuan Chen[†], Yue Cheng
George Mason University, [†]*JD.com Silicon Valley R&D Center*
{jzhang33, awang24}@gmu.edu, {min.li, yuan.chen}@jd.com, {yuecheng}@gmu.edu

1 Introduction & Motivation

More recently, a new paradigm called serverless computing (Function-as-a-Service or FaaS) has become prevalent, thanks to container-based virtualization. Serverless computing enables a brand new way of building and scaling applications and services by breaking the traditionally monolithic server-based application model into finer-grained functions; developers can thus focus on the development of function logics without having to worry about the server or VM management. While still in its infancy, extant serverless computing infrastructure starts to pose a set of issues. The foremost issue is the significant runtime overhead that hampers its elasticity and scalability. Simple adoption of container engines such as Docker [4] incurs significant container startup and runtime environment setup cost, thus making the serverless platforms fail to meet the latency and throughput requirement of bursty workloads.

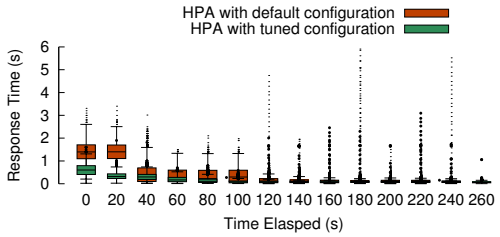

Figure 1: Careful HPA parameter tuning yields effective performance improvement and agile response to a workload burst.

Issues State-of-the-art open-source FaaS platforms [5, 6, 7] heavily rely on container orchestration frameworks such as Kubernetes [3] for container resource scheduling, which is originally designed for managing long-running applications [9] such as Memcached. FaaS-based applications demand elastic auto-scaling when workloads are bursty, whereas Kubernetes' container scheduling and auto-scaling protocol is too heavyweight to realize instant resource provisioning. For example, Kubernetes' horizontal pod autoscaler (HPA) first waits for multiple rounds of statistics collection until the measured metric (e.g., CPU usage of a certain pod) stabilizes before enters into the container scaling-out phase where the number of needed containers is calculated and provisioned. To demonstrate, we run HPA-managed OpenFaaS on a cluster of AWS EC2 VMs and issue a highly-concurrent function workload. As shown in Figure 1, the workload sees reduced response time after 40 seconds and stabilizes after 120 seconds, which is unacceptably long for a short-lived bursty workload. By carefully tuning the HPA parameters, we observe dramatically improved elasticity and quickly-converged function response time (the response time stabilizes after 20 seconds). This observation is because reconfigured HPA is more sensible to workload burst, and therefore, makes auto-scaling decision in a more timely fashion.

2 Designing HyperFaaS

Optimization such as container caching (or pre-warming, by launching a pool of empty standby containers ready for serving function requests) can not fundamentally solve the problem, because bursty workloads can exhaust the container cache pool quickly, resulting in excessive container provisioning afterward. HPA parameter tuning, on the other hand, can marginally mitigate the impact of slow auto-scaling to some extent. However, it pinpoints the root causes of such deficiency and motivates us to rethink the FaaS platform design. To this end, we propose HyperFaaS—a redesign of the FaaS platform that supports truly elastic resource scaling with high resource efficiency. The goal of HyperFaaS is two-fold: (1) to maximize the resource utilization and efficiency through hierarchical scheduling and container sharing; and (2) to minimize performance loss due to highly-concurrent bursty function workloads via transient resource scaling-up.

Hierarchical Scheduling Within one container, there are two management modules: (1) an in-container scheduler (ICS) that routes requests to (2) function workers that serve the function requests. Each ICS performs periodic metrics monitoring and proactively exchanges information with a centralized global scheduler (GS). The ICS keeps the quality-of-service (QoS) of function request serving in check with pre-defined resource (CPU/memory) usage watermarks. Whenever the

[*]Jingyuan and Ao are graduate students at GMU.

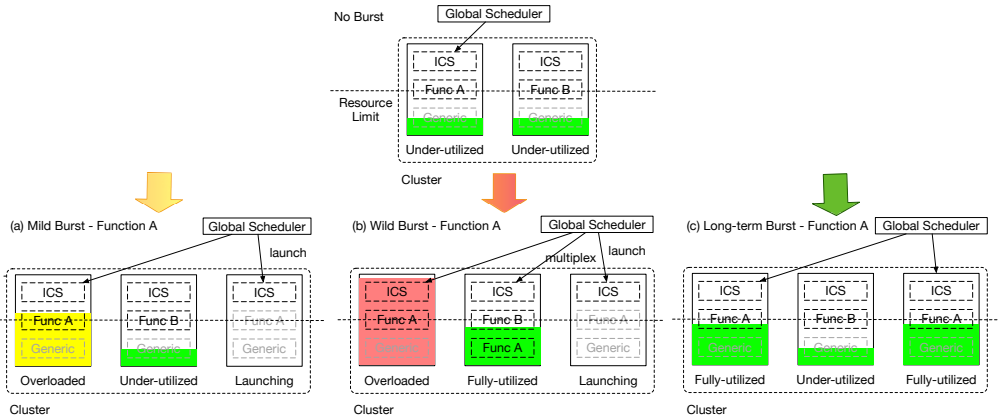

Figure 2: Overview of multi-phase auto-scaling scheme: (a): Transient scaling-up offers a temporary solution to mitigate the impact of mild burst, while waiting for the new containers to be launched from background; (b): Container multiplexing effectively serves wilder burst; (c): When the scaling-out process is finished, distribute the persisting burst to the new containers.

high watermark is reached, ICS notifies GS to take actions. Additional requests that would have overwhelmed that particular container will be bounced back to the GS, which performs rescheduling by looking for a different set of containers with available resources. This hierarchical scheduling scheme can effectively prevent container overwhelming and thus significantly improve the average and tail latencies.

Multi-phase Auto-scaling To effectively handle suddenly bursty workloads, we design a multi-phase "burst-buffer" styled scaling mechanism, where we follow a "scaling-up then scaling-out" design choice. A phase-one resource scaling-up is always applied first, whenever a bursty workload comes, with the hope that the burst is ephemeral and short-lived, as shown in Figure 2(a); HyperFaaS triggers the scaling-out as a last resort if the workload spike persists. Our phase-two scaling-out policy coordinates with GS to try to locate containers with available resources to sustain the increased workload, (Figure 2(b)); if there are not enough free resources, HyperFaaS enters into phase-three by asynchronously spinning up (via cold-start) more containers in the background, while phase-one scaling-up is used as a burst buffer to help form a smooth latency decrease curve (instead of a sudden jump suffered by existing platforms), as shown in Figure 2(c).

Container Sharing To further complement the above design decisions, HyperFaaS implements container sharing through a multi-function based resource multiplexing, with the goal of improving resource utilization. For this purpose, we adopt a double-buffering typed approach, where two function workers are co-existing within one container. Our container sharing scheme features a carefully-designed and lightweight function switching protocol that works as follows: when a container is lightly-loaded, one of two workers is loaded with function codes serving requests, while the other worker stays generic without being specialized; workers can switch between generic (unspecialized) and non-generic (specialized with source codes of the specific function). When a burst goes beyond the scaling-up threshold, GS will locate containers that are currently under-utilized, and specialize the generic worker for container multiplexing.

Implementation in Progress We are in the progress of implementing a prototype of HyperFaaS. To minimize the overhead of ICS, we use request streaming and directly pipe data to the worker without parsing the HTTP headers. The hierarchical scheduling and container sharing substrates are under heavy development. For the evaluation purpose, only CPU and memory resource will be considered in the first version.

3 Related Work

Extant serverless platforms [2] suffer from containers' cold startup and use container caching for performance improvement. SOCK [8] proposes a serverless-optimized container engine to mitigate the impact of long container latencies. SAND [1] introduces per-workflow container sharing to optimize performance for workflow-based serverless workloads. The proposed work shares with these systems the goal of reducing the container-level costs, but differs in its focus on resource scheduling.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: towards high-performance serverless computing. In *USENIX ATC*, 2018.

[2] Amazon. Aws lambda. https://aws.amazon.com/lambda/.

[3] The Kubernetes Authors. Production-grade container orchestration - kubernetes. https://kubernetes.io/.

[4] Docker. Enterprise container platform. https://www.docker.com/.

[5] Alex Ellis. Introducing functions as a service. https://goo.gl/c8mnD2.

[6] Fission. Serverless functions for kubernetes. https://goo.gl/VJKbGV.

[7] Kubeless. Kubeless. https://kubeless.io/.

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

[9] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Eurosys*, page 18. ACM, 2015.